Documentation and conceptual development of software components for the execution of geometric Boolean set operations on the basis of Java3D

Michael Theiler

Bauhaus-Universität Weimar
Fakultät Bauingenieurwesen
Professur Informatik im Bauwesen

May 05, 2010

Bauhaus-Universität Weimar
Fakultät Bauingenieurwesen
Professur Informatik im Bauwesen

S t u d i e n a r b e i t

Thema:	Documentation and conceptual development of software components for the execution of geometric Boolean set operations on the basis of Java3D
eingereicht von:	Michael Theiler
Fach:	Grundlagen Graphischer Nutzeroberflächen
geb. am: in:	18.10.1985 Saalfeld / Saale
Seminargruppe:	BM/08
Matrikelnummer:	42179
Erstprüfer:	Prof. Dr.-Ing. K. Beucke
Zweitprüfer:	Dipl.-Ing. E. Tauscher Dipl.-Ing. J. Tulke
Bearbeitungsdauer:	6 Wochen
Ausgabedatum:	24.03.2010
Abgabedatum:	05.05.2010
Note:	

Weimar, 7. März 2010 Prof. Dr.-Ing. K. Beucke

Conceptual Formulation

Subject:

Documentation and conceptual development of software components for the execution of geometric Boolean set operations on the basis of *Java3D*

Context:

Complex buildings and other structures are cumulatively planned with software that supports the export of building information in the *STEP*-format on the basis of the *IFC* (*Industry Foundation Classes*). Because of the availability of this interface, it is possible to use the data of a building for further processing.

Within the *IFC*, several geometrical models for the visualization of building elements are provided. Among others, geometric Boolean set operations are needed to "subtract" openings from building elements (e.g. for windows or doors) - *CSG* (*Constructive Solid Geometry*).

Therefore, software components based on the algorithms [Laidlaw86[1]] and [Hubbard90[2]] were developed at the professorship *Informatik im Bauwesen* that support these functionalities on the basis of *Java3D*. However, it turned out in praxis, that these components are numerically instable and that there is no acceptable robustness or tolerance of errors. This is caused by mistakes in the implementation (bugs) as well as the insufficient handling of numerical inaccuracies. Further, a verification and, where applicable, a correction of qualitative substandard initial data is missing.

Prior to this student research project, the implementation of a self-contained application for a visual error control was initiated. This tool visualizes several program steps and their corresponding data. With use of this tool, the implemented algorithms can be analyzed in detail.

Task:

The papers [Laidlaw86[1]] and [Hubbard90[2]] are unsatisfactory describing some essential steps of the algorithm as well as implementation details to execute Boolean set operations on the basis of a *B-rep* (*Boundary Representation*) model. Hence, the algorithm should be documented comprehensible with the help of figures and pseudo code. Moreover, problems within the existing implementation shall be identified and possible solution strategies shall be provided.

---

[1] Laidlaw, D. H.; Trumbore, W.B.; Hughes J.F. (1986). “Constructive Solid Geometry for Polyhedral Objects”, Proceedings of SIGGRAPH ’86, published as Computer Graphics, Vol. 2, ACM, New York, USA

[2] Hubbard, P.M. (1990). „Constructive Solid Geometry for Triangulated Polyhedra“, Department of Computer Science, Brown University, Providence, Rhode Island 02912, CS-90-07, September 1, 1990

The following points have to be treated:

1. Detect critical points regarded to the mentioned numerical inaccuracies and the stability problems within the algorithms!
2. Which existing strategies have to be applied to reduce the problems in point 1? Read up on technical literature for this!
3. Which corrections of the initial data will increase the robustness of the software components?
4. Extend the existing debugging application and the implemented algorithms, where required, to test your solving strategies for success.

Organizational hints:

An intermediate guidance about the progress of the student research project at the professorship *Informatik im Bauwesen* is explicitly welcome. Weekly consultations have to be attended.

Table of Contents

1 Introduction

"The robustness of Boolean operations between solids is crucial for the usability of a solid modeler. Unfortunately, geometric modeling is like shoveling sand. With every shovel you pick up a bit of dirt. The numerically imperfect nature of geometric algorithms can challenge the Boolean engine with contradictions and inconsistencies." (P.H. Ernst [11], p.74)

The data exchange of construction plans between software of all kind of planners is a problem since plans have been designed with the computer. On account of this the *buildingSMART e.V.* (former *IAI*) has developed an open exchange standard for the building industry called *IFC* (*Industry Foundation Classes*). With this standard, all data from a building information model shall be described in one central model. The *IFC* are registered under the *ISO 16739* and have become more and more an accepted exchange format between different kinds of planning software.

Within several research projects at the professorship *Informatik im Bauwesen*, software components in *Java* were developed, that can be used for further processing of the data from an *IFC* file. These components are open source and reachable under *www.openifctools.org*.

The *IFC* provides several geometrical models to visualize building elements. One of these geometric models uses Boolean set operations on two or more solids, for instance, to "subtract" an opening element from another building element. For this purpose, a Boolean modeler was developed, because until now there is no complimentary software available in *Java* that can do this in a satisfactory manner. However, it turned out in praxis, that these components are numerically instable and that there is no acceptable robustness or tolerance of errors. This is caused by mistakes in the implementation (bugs) as well as the insufficient handling of numerical inaccuracies. Also a verification and, where applicable, a correction of qualitative substandard initial data is missing in these tools.

The developed Boolean modeler is written in *Java* and uses the add-on *Java3D*. It is based on the algorithms of *Hubbard* [1] and *Laidlaw et al.* [2]. These papers describe the strategies used to implement a Boolean modeler. Unfortunately, some essential steps are pictured unsatisfactory. Hence, in this paper the implementation details on the basis of a triangulated *B-rep* (*Boundary Representation*) model will be documented. Every step of the implementation will be explained in detail with the help of figures and, if suggestive, with pseudo code.

Within the second chapter, you will get a first insight about the basic functionalities of a Boolean modeler. Afterwards, within the third chapter, the used models will be presented. On the one hand, there is the initial data in the form of a *Java3D* geometry model, which is created before by analyzing the data of an *IFC* building information model. On the other hand, a more complex model based on *Hubbard* is needed to perform Boolean set operations.

A conversion between these models will be shown, that has to be accomplished before and after an operation. Next, in the fourth chapter all the needed algorithms and methods will be documented in detail.

Additionally, an extension of the *Hubbard* algorithm will be presented that allows a simultaneous computation of Boolean set operations with more than two solids. This extension was developed, because building elements within an *IFC* model are often the result of more than just one Boolean set operation. The execution time using this improved algorithm will be reduced, since much less computations have to be performed. Furthermore, an adaption of the *Hubbard* algorithm will be presented in order to clip a solid at a specified plane.

Moreover, in the fifth chapter, some problems of the current implementation will be discussed and, if applicable, a strategy to solve the problems will be shown. A validation of the initial data will be presented as well as some approaches that show how to increase the robustness of geometric algorithms like Boolean set operations. It will be estimated how far these approaches can increase the robustness of the presented software components.

Finally, the sixth chapter concludes the results of this work and gives an outlook on how the development of the software components shall be continued.

2 Functionality of a Boolean Modeler

The basis functionality of a Boolean modeler contains the possibility to get the result of the three basic Boolean set operations. This can be the difference (\), the union (∪) or the intersection (∩) of two solids. Beyond that, the software components, developed at the professorship *Informatik im Bauwesen*, need further functionality. All operations use the same algorithms and differ in just a few points. This paper will come back later on these issues. The following figures illustrate the different kinds of Boolean set operations that have been implemented:

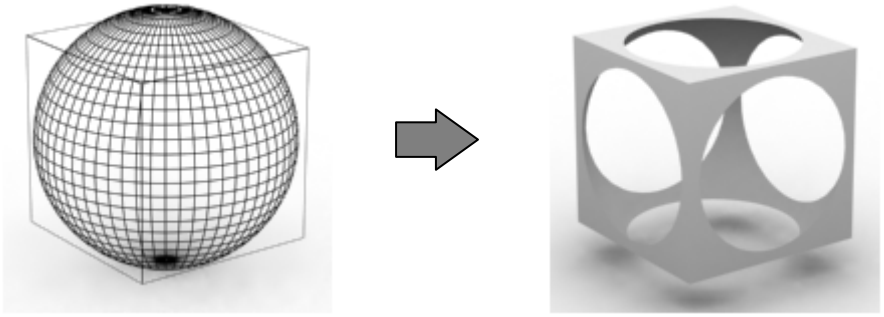


Figure 2.1 - Example: Boolean difference (cube \ sphere) [5]

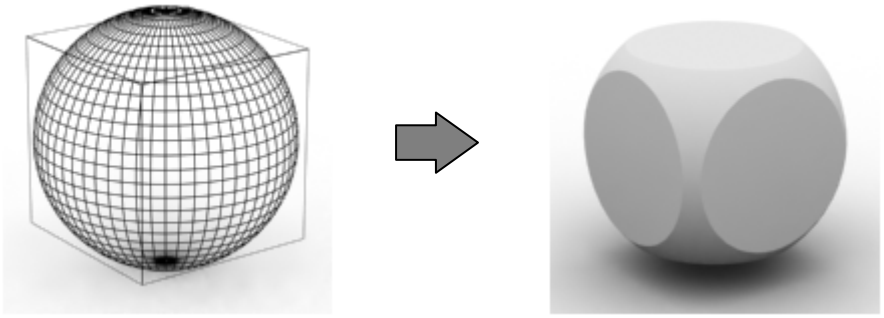


Figure 2.2 - Example: Boolean intersection (cube ∩ sphere) [5]

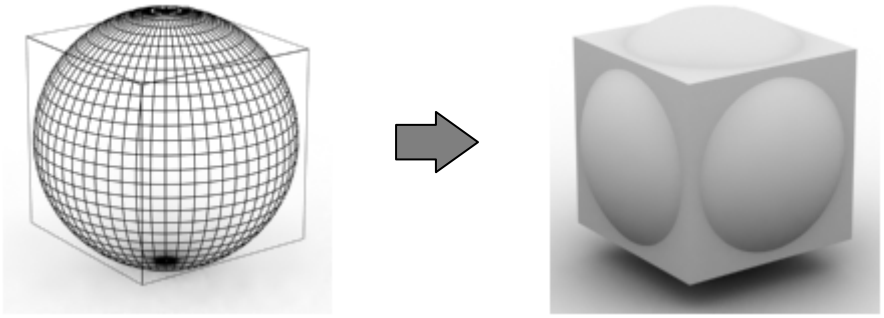


Figure 2.3 - Example: Boolean union (cube ∪ sphere) [5]

further functionality:

- *decomposing*

  This procedure computes the decomposition of a solid by a specified zone expressed as polyhedron. Solids will be divided into parts that are inside or outside the zone. The procedure's implementation details are not part of this paper - see [4] for more information.

- *clipping*

  This procedure computes the result of a solid that is clipped at a specified unbounded plane.

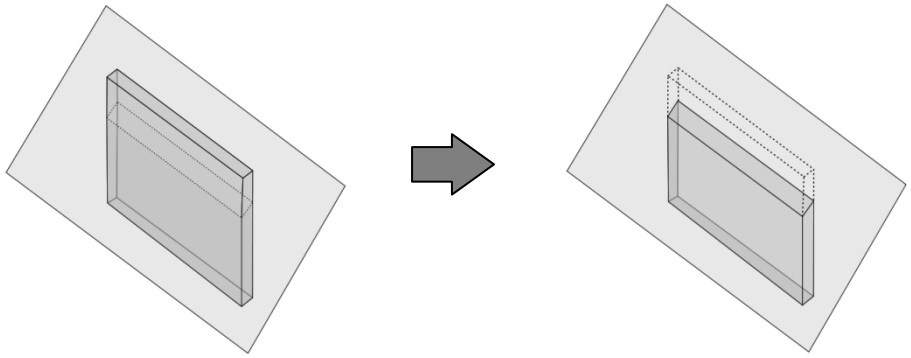


Figure 2.4 - Example: Clipping at a specified plane (according to [6], p.136)

- *polygonal bounded clipping*

  This procedure computes the result of a solid that is clipped at a specified plane. Unlike to the previous procedure, the half space is limited by a polygonal boundary as can be seen in *Figure 2.5*.

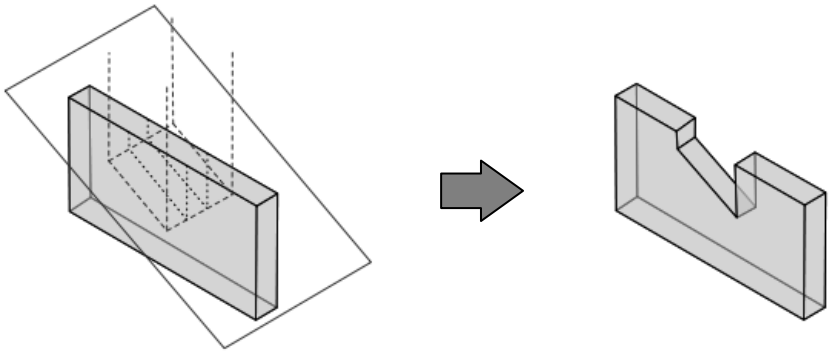


Figure 2.5 - Example: Polygonal bounded clipping at a specified plane (according to [6], p.50)

3 Model Abstraction

This paper focuses on the computation of Boolean set operations of triangulated solids based on a *B-rep* geometry model. An implementation is presented based on the algorithm of *Hubbard* [1], which restricts and improves the algorithm of *Laidlaw et al.* [2] to process the special case of triangulated polyhedra. The *Laidlaw et al.* algorithm was originally designed for any polyhedron and has a poor performance, if triangulated surfaces are used. Its main weakness is the purely local approach used by its splitting phase. Thereby, too many triangles will be created as it is shown in the paper of *Hubbard* as well as later in this paper.

This chapter summarizes the abstraction of the *Hubbard* model and presents its implementation within the developed Boolean modeler. It will be shown how the initial data from the building model's visual representation can be stored using the *Java3D* model. After a short abstract about the *Java3D* geometry model, the conversion between the *Hubbard* and *Java3D* model will be presented.

3.1 The Hubbard Model

"Triangulated polyhedra offer a number of advantages over general polyhedra. The data structures that represent a general polyhedron are complicated by the fact that each face can involve an arbitrary number of vertices and edges. This problem does not arise in the data structures for a triangulated polyhedron, since each of its faces is defined by exactly three vertices and edges." (P.M. Hubbard [1], p.3)

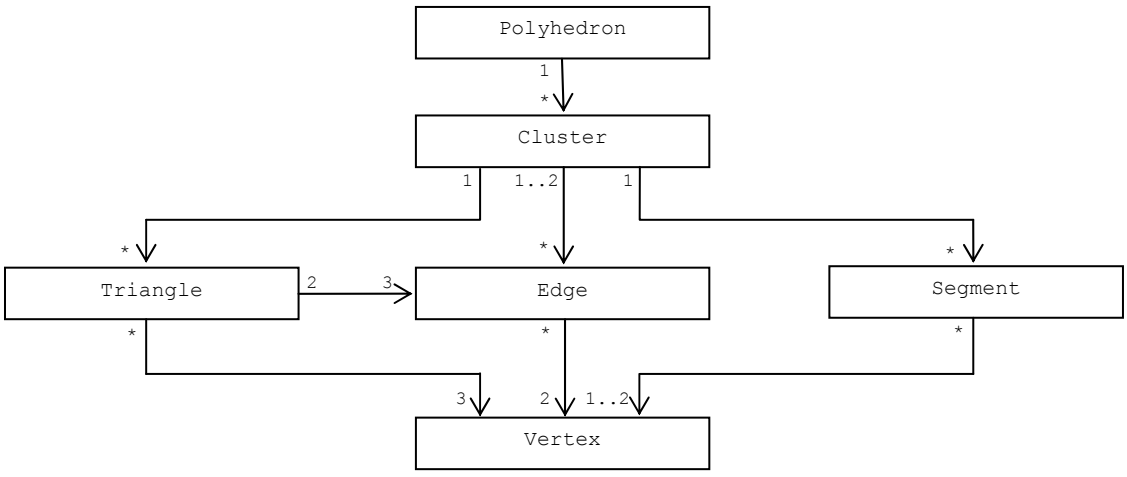


Figure 3.1 - Class structure of the Hubbard model

The purely local approach used by the splitting phase of the *Laidlaw et al.* algorithm is one of its main weaknesses. To solve this problem, *Hubbard* introduces a global processing of intersections. Therefore, all triangles of a polyhedron will be summarized to clusters. These

clusters are planar surfaces, which will be used to implement a more global splitting phase after all intersections have been found. For example, the simplest representation of a cube as triangulated polyhedron consists of twelve triangles. Two triangles can always be summarized to a cluster, since these pairs of triangles are adjacent faces and share the same normal vector.

A cluster may have a plenty of triangles, but a triangle is always connected to only one cluster. The triangle structure has references to its three vertices and to the three edge objects connecting them. The edge structure contains references to the two vertices at its endpoints and to the two triangles adjoining it (*Figure 3.1*). Edges will be classified as *exterior edges*, if they separate two clusters. Otherwise, if they just separate two triangles within the same cluster, they will be called *interior edges*. Further, the *Segment* class is used to store an intersection segment *s* between triangles (*Figure 3.2*). This intersection can either be a line or just a single point, provided that these triangles are intersecting against each other and that they are not coplanar. The coordinate of a segment's vertex is stored by its distance (e.g. *d1* or *d2*) to a reference point *RP* on the intersection line *L* of the triangle's planes.

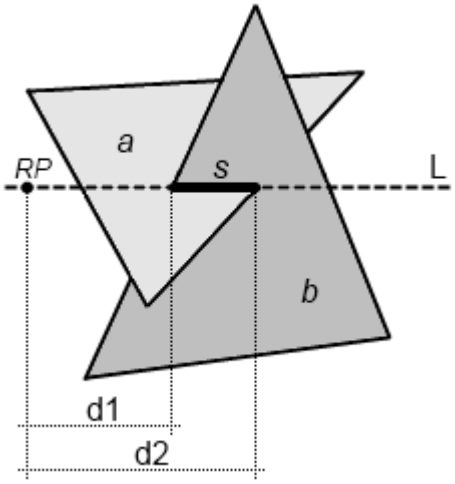
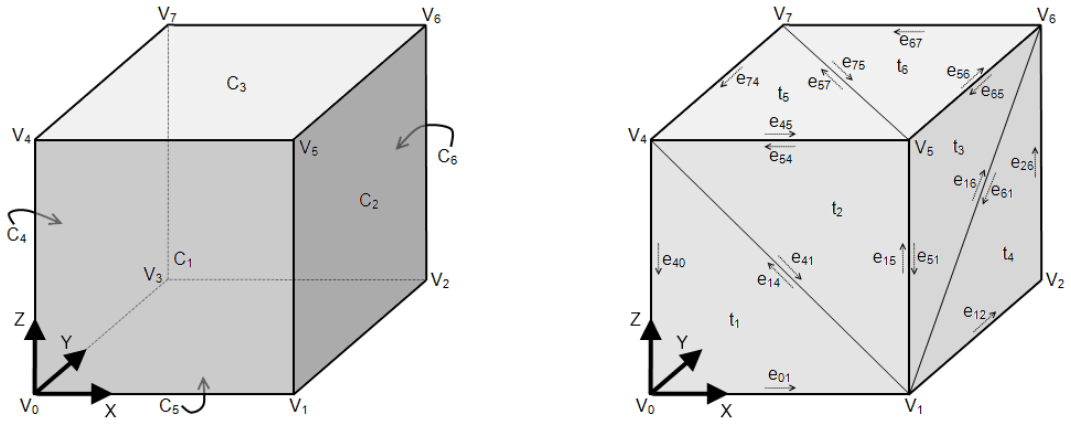


Figure 3.2 - Two triangles *a* and *b*, their intersection segment *s* and the distances *d1* and *d2* of the segment's vertices to a reference point *RP* on the intersection line *L*

3.2 Extending the Hubbard Model

The data model of the implemented software components differs in some points from the *Hubbard* model or even extends it. The most important difference concerns the geometrical data model. The *Hubbard* model uses a structure, where an edge is shared by two triangles. This will breed edges that belong to only one cluster (*interior edges*) and edges that belong to two clusters (*exterior edges*). In contrast, the data structure of the implemented components is based on an object oriented variant of the data structure developed by *Muller and Preparata* [4, 5], which they call *doubly connected edge list* (*DCEL*). *Figure 3.3* shows an example of such a *DCEL* structure with triangles. In principle each edge exists twice - one edge for each of two adjacent triangles. This also means that one edge is explicitly associated

to one cluster contrariwise to the original *Hubbard* model. The edges are treated as directed edges. In this way, each edge in a 2-manifold solid must have a *twin edge*, which is just inverted in its direction - but still connecting the same vertices. For instance, if you look at edge e_{14}, you will find its *twin edge* e_{41}. You will also see that these edges belong to different triangles: Edge e_{14} is connected to triangle t_1 and edge e_{41} to triangle t_2. The edges of a triangle are always oriented counter-clockwise.



Polyhedra:

Polyh.	*Clusters*
cube	{C_1, C_2, C_3, C_4, C_5, C_6}

Clusters:

Cl.	*Δ*
C_1	{t_1, t_2}
C_2	{t_3, t_4}
C_3	{t_5, t_6}
C_4	{t_7, t_8}
C_5	{t_9, t_{10}}
C_6	{t_{11}, t_{12}}

Triangles:

Δ	*Edges*
t_1	{e_{01}, e_{14}, e_{40}}
t_2	{e_{41}, e_{15}, e_{54}}
t_3	{e_{51}, e_{16}, e_{65}}
t_4	{e_{61}, e_{12}, e_{26}}
t_5	{e_{45}, e_{57}, e_{74}}
t_6	{e_{75}, e_{56}, e_{67}}
…	

Edges:

Edge	*from*	*to*	*Δ*
e_{01}	V_0	V_1	t_1
e_{14}	V_1	V_4	t_1
e_{40}	V_4	V_0	t_1
e_{41}	V_4	V_1	t_2
e_{15}	V_1	V_5	t_2
e_{54}	V_5	V_4	t_2
e_{51}	V_5	V_1	t_3
e_{16}	V_1	V_6	t_3
e_{65}	V_6	V_5	t_3
e_{12}	V_1	V_2	t_4
e_{26}	V_2	V_6	t_4
e_{61}	V_6	V_1	t_4
e_{45}	V_4	V_5	t_5
…			

Vertices:

Vertex	*Coord.*
V_0	(0, 0, 0)
V_1	(1, 0, 0)
V_2	(1, 1, 0)
V_3	(0, 1, 0)
V_4	(0, 0, 1)
V_5	(1, 0, 1)
V_6	(1, 1, 1)
V_7	(0, 1, 1)

Figure 3.3 - Example: Extended Hubbard model of a cube using a doubly connected edge list (DCEL)

Internally all vertices will get a unique number for their distinct identification. The vertex objects and the corresponding numbers or indices are stored in a map, which organizes the internal numbering of the vertices. This coordinate map insists of two hash maps - one for mapping the index to the specified vertex object and the other one for the inverse relation. Beside this, it contains a tree set of vertex objects in order to avoid adding a vertex twice within a certain tolerance. Because of the data structure of a tree set, an existing vertex in the set can be found very quickly (binary search) by using a comparator for the x-, y- and z-coordinates.

The class structure of the extended *Hubbard* model is illustrated in *Figure 3.4*. A vertex object is represented by the *Java3D* class *Point3d*. Every class illustrated in this figure has

access to the mentioned global coordinate map, which manages the index coordinate relation as previously described.

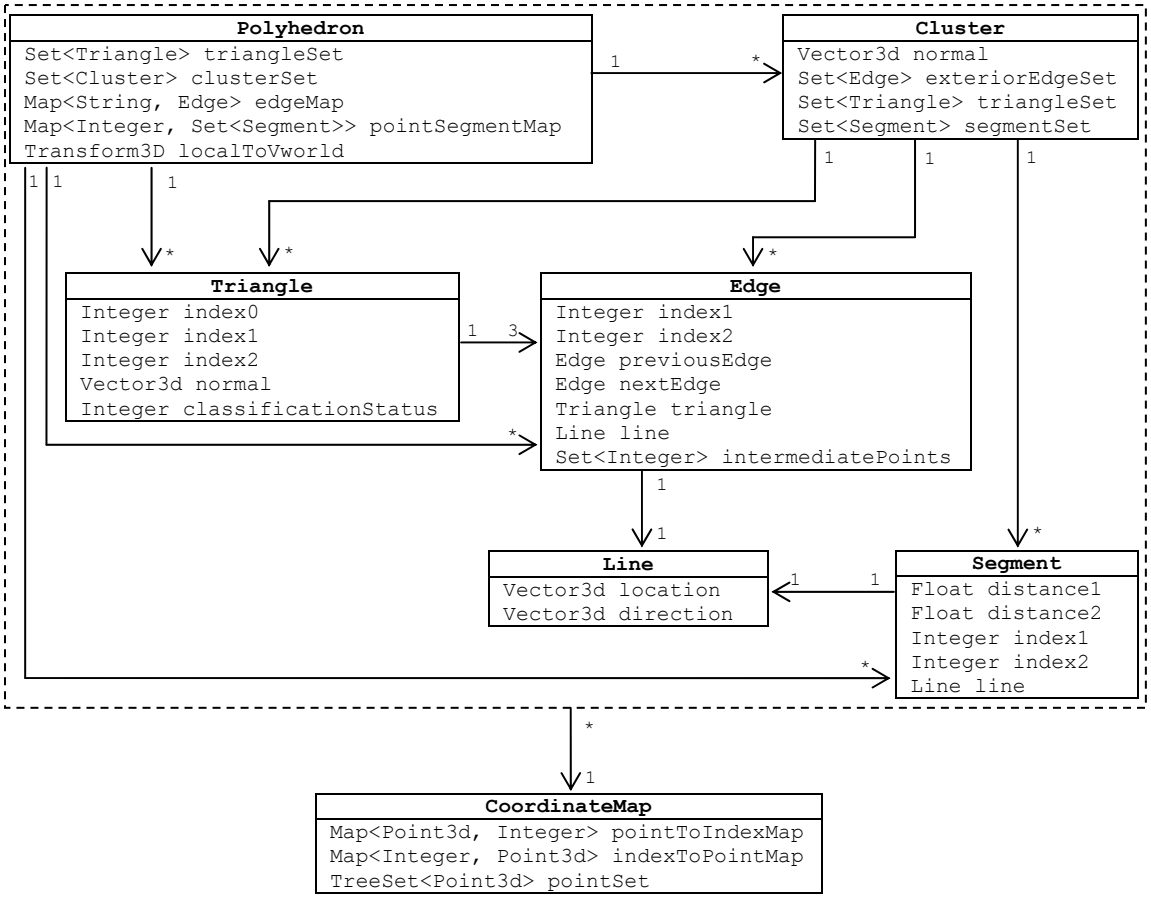


Figure 3.4 - Class diagram of the extended Hubbard model

Polyhedron: A polyhedron is divided into clusters as in the original *Hubbard* model, but it also has references to all triangles and edges. This is caused by the fact that cluster objects do not exist during the instantiation of a polyhedron object - they have to be generated at a later date. Moreover, the polyhedron class contains a map storing segments located at a specific vertex. This is needed for locating segments at a specified vertex within the intersection segment optimization algorithm, which will be documented later within this paper. A 4x4 matrix stores the transformation from the building element's local coordinate system to the building model's global coordinate system. This matrix is needed to transform the vertices during the model conversion process. This will be discussed within the next chapters.

Edge: The end points of edges are stored via the index number of their vertices in the global coordinate map. Furthermore, an edge has a direct reference to its triangle object and to its *next* and *previous edge* object within this triangle. The *next edge* is defined as the edge, which starts at the end point of the considered edge. Contrariwise, the *previous edge* ends at the start point of the considered edge (*Figure 3.5*). These attributes are needed for the navigation within the polyhedron's topological structure. Furthermore, intermediate points on

exterior edges have to be stored. These points will be created during the intersection phase and will be needed for splitting the edges.

In order to get quick access to the edge objects of a polyhedron, edges are mapped according to their name. Edges are named by the index numbers of their aligned vertex objects separated by a comma. For example, if you want to have the edge object of e_{41} from the example above, you will get it by accessing the polyhedron's edge map using the key *"4,1"*. This brings up the advantage that a *twin edge* does not need to exist at the edge object's instantiation time. Anyhow, it is possible to "ask" an edge for its *twin edge* after the instantiation process of all edges is completed. At this point, the edge will return the object mapped by the inverse key *"1,4"*, which has to exist in a regular 2-manifold solid with a valid triangle mesh.

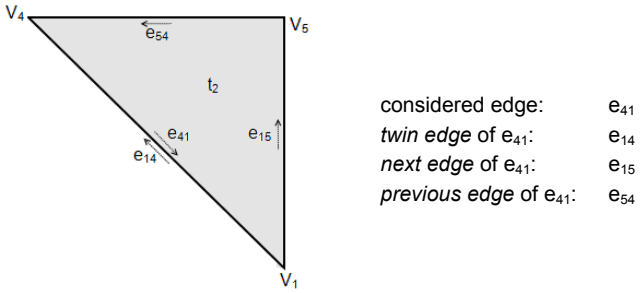


Figure 3.5 - Denotation of edges

Triangle: The *Triangle* class does not store its vertices itself; it just has three integer values representing the corresponding vertex's index within the coordinate map - as it is implemented in the *Edge* class. Beyond that, there is a reference to the triangle's normal that is computed once during the triangle's instantiation phase. The classification status is stored as an integer value. It describes how a triangle is classified (inside, outside or on the surface of the other polyhedron or even unclassified). The triangle's edge objects can be accessed via the polyhedron's edge map using the indices of two vertices as key, as mentioned earlier.

Segment: The *Segment* class stores its vertices via the distances to a reference point on the triangle's intersection line as described in the original Hubbard model (*Figure 3.2*). Furthermore, the indices of the end vertices are stored, after the segment's creation phase is finished. A difference to the original *Hubbard* model is that only line segments will be regarded. Point segments will be neglected, because they are not necessary for the further decomposition of the surfaces.

Line: Both, the *Segment* and the *Edge* class have a reference to a *Line* object that represents an unbounded line, where the objects are located on. It contains a location vector as well as a direction vector. The location vector is used to store a segment's reference point on the intersection line as mentioned earlier.

3.3 The Java3D Model

Java3D provides several methods for an efficient handling and rendering of triangulated geometry as well as several options to store a solid's geometrical structure. One efficient method is called *Indexed Triangle Array* (see also *Java3D Tutorial* [7], pp. 2.20-2.34). Every other geometry can easily be converted to this format using appropriate methods of *Java3D*. Using this format, the vertices of a solid are indicated with the effect that multiple used vertices only exist once. The list of coordinates, used in this geometry model, is stored within the *coordinate data array*. A tolerance checking for the determination of the equality of vertices is not used. Further, the *coordinate index array* contains the topological structure describing the solid's triangles. Three successive indices of this array represent the vertices of a triangle as it is illustrated in *Figure 3.6*.

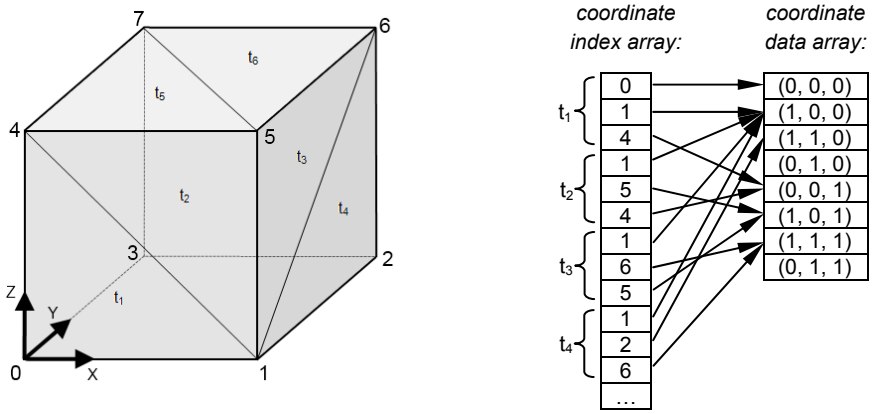


Figure 3.6 - Example: Indexed Triangle Array geometry

Solids in *Java3D* are represented by the class *Shape3D*. This class contains the geometry (for instance the mentioned *indexed triangle array* geometry) as well as an appearance object that contains the visual rendering information. The *Shape3D* objects are arranged in a *scene graph* that has a tree structure. This *scene graph* can contain several kinds of nodes, for example for grouping other nodes (*BranchGroup*), changing the visibility of all children below a node (*Switch*) or transforming all children below a node (*TransformGroup*) - for more information see [7, 22].

A *TransformGroup* gives you the possibility to define local coordinate systems via transform matrices. A similar approach is used within the *IFC* model in order to define local coordinate systems. For example, if you imagine a building, then every storey may have a local coordinate system. The geometry of the building elements itself are possibly denoted again in a local coordinate system (*Figure 3.7*). Therefore, if you want to compute the result of a Boolean set operation of an element from the first storey with another element from the second storey, you have to get the correct position of both solids in the building model's global coordinate system. All transformation matrices above the *Shape3D* node in the *scene graph* have to be composed. This can be done by calling the method *getLocalToVworld()* of the *Shape3D* class. This method returns a 4x4 matrix with the composite of all transforms in

the *scene graph* from the root down to this node (in the following called *local to virtual world matrix*).

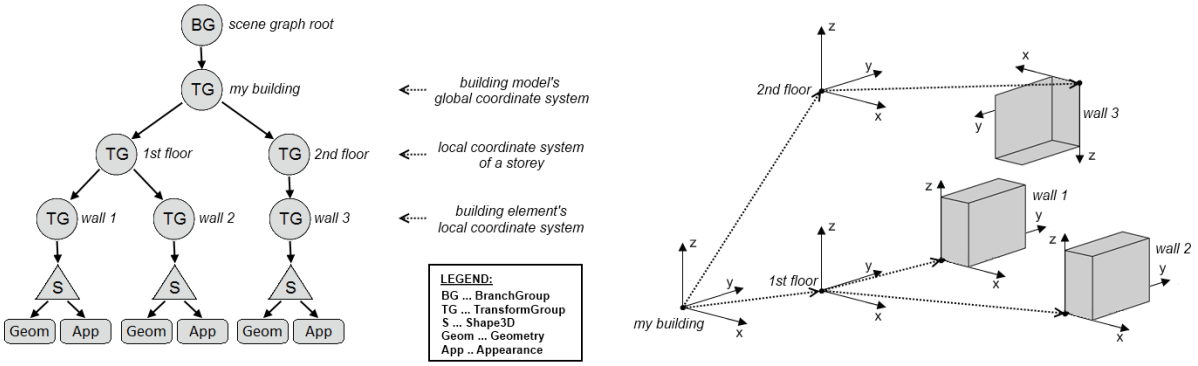


Figure 3.7 - Example: Relative positioning of local coordinate systems within a *scene graph*

3.4 Conversion between the Models

Both, the *Java3D* model and the extended *Hubbard* model, are *B-rep* geometry models and use an indexed coordinate structure. (At least the presented *indexed triangle array* format is an indexed geometry. Some geometry formats of *Java3D* follow up other strategies). The *Java3D* model is not that extensive than the *Hubbard* model, since just the geometry has to be stored in an efficient way for a solid's visualization. In contrast, the *Hubbard* model contains much more information about a solid's topological structure, for example the affiliation of triangles to clusters.

At the beginning of Boolean set operation the *Java3D* model has to be converted to the extended *Hubbard* model. After the operation is completed, the result that is computed has to be transferred back to the *Java3D* model. This section will give you an overview, which steps have to be treated during the conversion process in both directions.

At the beginning, the data structure of the extended *Hubbard* model has to be created based on the data of the *Java3D* geometry model. If the *Java3D* geometry is not yet in the *indexed triangle array* format, it has to be converted using internal methods of *Java3D*.

In a first step, each solid has to be transformed into the global coordinate system. Therefore, the *coordinate data array* of the indexed geometry will be read out and each vertex has to be multiplied with the *local to virtual world matrix* of the respective *Shape3D* object. This has to be done to make sure that every *Shape3D* object is located in the same coordinate system. Next, the coordinates can be copied to the global coordinate map of the extended *Hubbard* model. The index numbers of the coordinate map have to be consistent with the index numbers of the indexed geometry array, because in the next step the triangles' coordinates, that are stored in the *coordinate index array,* will be used to create the topological structure on the basis of these indices. Always three index elements in this array

result in one triangle. Consequently, the number of triangles of a solid is equal to the length of the *coordinate index array* divided by three.

After instantiating a triangle object, its normal has to be computed once. Therefore, one vertex has to be selected and the vectors from that vertex to the other two vertices can be computed. The normal results in the cross product of these vectors. Further, each created triangle demands a generation of its three edge objects. All created triangle and edge objects have to be referenced in the data structures of the polyhedron. These steps have to be repeated for each solid of the current Boolean operation. Afterwards, the generation of the polyhedron's clusters can be performed as it is described later in this paper.

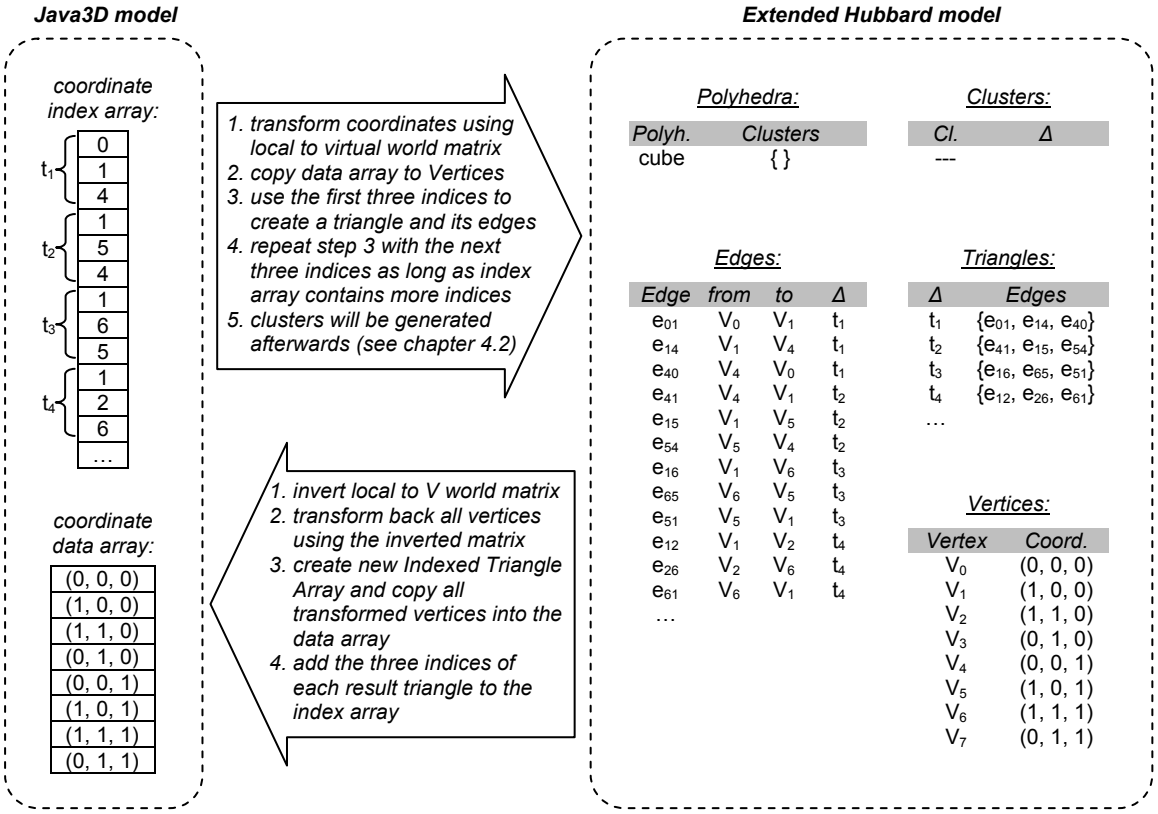


Figure 3.8 - Conversion between the models

At the end of all processing the result of the Boolean set operation has to be returned. Therefore, the information from the extended *Hubbard* model has to be converted back to the *Java3D* geometry model. In a first step, the *local to virtual world matrix* has to be inverted in order to transform back all coordinates into the local coordinate system of the initial solid. Next, a new *indexed triangle array* geometry can be created containing the indices of the result triangles and the corresponding coordinates. This geometry will then be used to create a new *Shape3D* object, which will be returned as the final result of the Boolean set operation.

4 Methods and Algorithms for Boolean Operations

The working method of a Boolean modeler according to the *Laidlaw et al.* algorithm [2] is to find the polygonal boundaries of the geometrical intersection of two polyhedral solids *A* and *B*. The faces bounded by these polygons have to be classified in each case by their location relative to the other solid. A face can be located inside, outside or on the boundary of the other solid.

Unfortunately, some faces may be partially inside and partially outside the other polyhedron (or even partially on the boundary). Because of this, it is not possible to classify those faces. For this reason, such faces have to be splitted first with the result that afterwards only faces remain, that are fully inside, outside or on the boundary of the other solid.

Depending on the operation that is executed, the algorithm decides which faces belong to the resulting solid and which faces can be neglected. Thus, it is not decisive, if faces are triangles or general polygons, but triangulated geometry is much easier to handle. As a consequence, there are simplifications possible within the algorithm, because many special cases can be excluded. Therefore, the algorithm described by *Hubbard* [1] restricts and optimizes the *Laidlaw et al.* algorithm [2] for the application with triangulated polyhedra.

The provided Boolean modeler implementation is based on the algorithm described by *Hubbard*. Some steps of the implementation differ from the suggested algorithms, due to the availability of better solution strategies. Other parts were developed on our own, because the paper of *Hubbard* is not precise in some cases.

The first section of this chapter gives you an overview over all implemented algorithms necessary to compute the result of a Boolean set operation. The following sections describe in detail the algorithms' work steps. Beyond this, *chapter 4.8* introduces an extension of the *Hubbard* algorithm to support a simultaneous computation of several Boolean set operations. Afterwards, an adaption is presented that allows the clipping of a solid at a plane.

4.1 Algorithm Overview

This section will give you an overview of the implemented algorithms. The algorithms are grouped to four phases - the *initial phase*, the *splitting phase*, the *classification phase* and the *finishing phase*. These phases can be broken down into sub problems as illustrated in *Figure 4.1*. In the following, the sub problems are introduced briefly. A more detailed description of an algorithm can be found in the respective chapter.

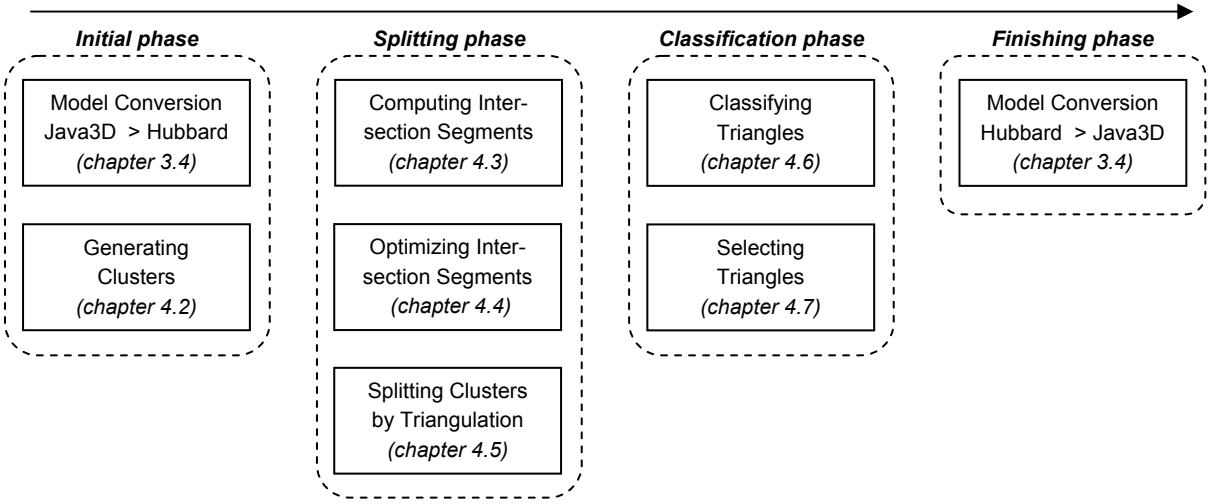


Figure 4.1 - Algorithm overview

Initial Phase: At the beginning the *Java3D* model has to be converted into the extended *Hubbard* model as it is described in *chapter 3.4*. Next, the cluster structure has to be generated, whereby all coplanar adjacent triangles will be composed to clusters - see *chapter 4.2*.

Splitting phase: Within the *splitting phase* the algorithm searches for intersection segments. Next, the clusters are splitted along these sections. In this manner, only triangles remain that are fully inside, outside or on the boundary of the respective other polyhedron. This phase can be subdivided in three steps:

> **Computing Intersection Segments:** Each triangle of *A* is tested against each triangle of *B* for intersection. A pretest detects quickly, if two triangles intersect. In the case of no intersection the processing can be stopped and the pretest can continue with the next triangles. Coplanar faces are not regarded, because it is sufficient to consider only non coplanar triangles. *Hubbard* explains very detailed, why the intersection of coplanar faces can be ignored (see [1], p.8). However, if two triangles intersect, an intersection segment will be created and stored as a constraint for splitting the clusters. The computation of this segment is explained in *chapter 4.3*.

Optimizing Intersection Segments: *Hubbard* depicts that the result of the segment computation has to be optimized, because only in this manner a significant reduction of triangles and, consequently, a reduction of processing time can be achieved in comparison to the original *Laidlaw et al.* algorithm. Unfortunately, this step is sketched very briefly within the *Hubbard* paper. For this reason, an own algorithm was developed in order to combine segments - see *chapter 4.4*.

Splitting Clusters by Triangulation: Finally, the clusters of the polyhedra have to be splitted along the computed and optimized intersection segments. The splitting is an instance of a planar constrained triangulation problem. The constraints conclude the *exterior edges* of a cluster as well as the intersection segments within this cluster. The triangulation step and special cases depending on the used triangulator are sketched in *chapter 4.5*.

Classification phase: The second phase classifies each triangle of *A* against *B* and vice versa as follows:

- *INSIDE*: The considered triangle is located fully inside the other polyhedron.
- *OUTSIDE*: The considered triangle is located fully outside the other polyhedron.
- *SAME*: The considered triangle is located on the boundary of the other polyhedron. The triangle's normal and the boundary's normal are pointing into the same half space.
- *OPPOSITE*: The considered triangle is located on the boundary of the other polyhedron. The triangle's normal and the boundary's normal are opposed.

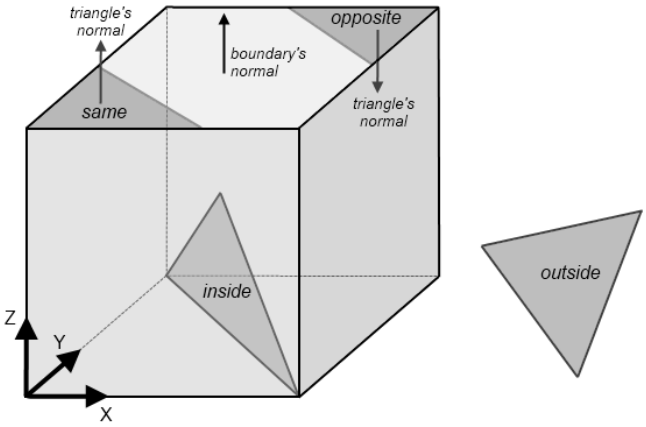


Figure 4.2 - Possible classifications

Classifying Triangles: The classification is computed by analyzing the constellation of triangles along a common intersection segment. Thereby, the constellation of a triangle from one polyhedron will be analyzed relative to two other triangles of the second polyhedron. Furthermore, triangles will be classified by propagating the classification status of a triangle to its neighbors according to certain rules. Ray

casting as it is used in the original *Hubbard et al.* algorithm is avoided except for some special cases. The classification and propagation algorithm is depicted in *chapter 4.6*.

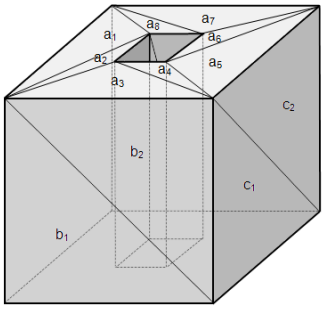
Selecting Triangles: As soon as all triangles of both polyhedra are classified, the result of the Boolean set operation can be composed easily by selecting the needed triangles depending on the operation that is performed. For example, if *A* and *B* should be united, the result will contain the triangles of *A* that are *outside B*, the triangles of *A* that are classified as *same* and the triangles of *B* that are *outside A*. An overview which triangles remain depending on the performed operation is given in *chapter 4.7*.

Finishing phase: Finally, the Hubbard model has to be converted back to the *Java3D* model. The *local to virtual world matrix* has to be inverted in order to transform back all coordinates. Next, the *Java3D* indexed triangle geometry has to be created as it is described in *chapter 3.4*.

4.2 Generating Clusters

This algorithm composes triangles of a polyhedral solid to clusters. It is developed by own, since *Hubbard* does not describe a method how to generate the cluster structure. Beginning with one arbitrary triangle, all neighbors of this triangle have to be checked for coplanarity. This can be tested by comparing the equality of the triangles' normals. If they are not equal within a certain tolerance, the edge between those triangles can be marked as an *exterior edge* of this cluster - else the start triangle and the current observed triangle can be composed to one cluster. The adjacent triangles of the current observed triangle have to be checked now, whether they can be added to this cluster, too, or not. This procedure has to be repeated as long as no other adjacent triangle can be detected, that has the same normal vector as the start triangle. In this case the current cluster is complete. The described process will be repeated with an arbitrary triangle that is not already allocated to a cluster. The algorithm terminates, if all triangles are dedicated to a cluster.

To show the algorithm's working method, it is demonstrated with help of the example solid of *Figure 4.3*. In a first step, a new cluster will be created and an arbitrary triangle has to be added to this cluster - for example a_3. Now, the normal vectors of the adjacent triangles a_2, a_4 and b_2 have to be tested for equality with the normal of a_3. This test shows, that a_2 and a_4 can be added to the cluster containing already a_3. Triangle b_2 is not coplanar, consequently, the edge between a_3 and b_2 is an *exterior edge* of the current cluster and b_2 cannot be added to this cluster. Afterwards, the adjacent triangles of a_2 and a_4 have to be tested. This test returns that there are further triangles that are coplanar to a_3. The algorithm continues repeating the process as long as no other adjacent coplanar triangle can be detected. Consequently, the first cluster contains the triangles a_1, a_2, a_3, ... , a_8. Now, a new cluster can be created containing another arbitrary triangle that is not associated to a cluster yet - for example b_1. The algorithm detects, that b_1 and b_2 can be composed. In the following steps, further clusters will be created containing c_1 and c_2 as well as the other not illustrated faces.



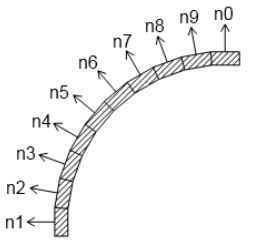
Clusters:
$C_1 = \{ a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8 \}$
$C_2 = \{ b_1, b_2 \}$
$C_3 = \{ c_1, c_2 \}$
…

Figure 4.3 - Example: Cluster generation

A clever navigation over the DCEL structure according to this strategy can be reached using the presented edge navigation. Thereby, the edges that were investigated will be marked as visited. If the algorithm comes back to such an edge, it is not necessary to test the triangles separated by this edge again. Because of the repetitive test of all adjacent triangles, a recursive implementation of the algorithm is suggestive (*Algorithm 4.1*).

Analogous to the described strategy above, the algorithm starts with an arbitrary edge. A new cluster will be created and the triangle associated with the current edge and the edge itself will be added to the appropriated sets of the cluster. The current edge has to be marked as visited. The normal of the cluster is set to the normal of this first triangle.

It is necessary to define a normal of a cluster to avoid adding triangles of a curved surface to a single cluster. For example, if you imagine a curved wall, then adjacent triangles on the curved surface may have equal normal vectors within a certain tolerance, which has to be used to eliminate numerical inaccuracies. As a consequence, if you just compare the normal vectors of two adjacent faces, you may get the result that all faces along this curved wall belong to one cluster (*Figure 4.4*). But this is obviously not correct. To avoid this, you always have to check the normal vector of the current triangle with the initial normal vector of the first triangle you choose (respective the normal of the cluster).



n1 ≈ n2
n2 ≈ n3
n3 ≈ n4
…
n8 ≈ n9
n9 ≈ n0

but n1 != n0

Figure 4.4 - Necessity of determining a cluster's normal that is used for the comparison of normals instead of comparing only the normals of adjacent triangles

The algorithm continues checking, whether the current cluster's *twin edge* was visited already or not. Of course, at the beginning no edge can be visited already, but possibly at a later date - remind the algorithm is recursive. If this applies and if the *twin edge* is marked as an *exterior edge*, the current edge is also an *exterior edge* and it has to be added to the current

cluster's set of exterior edges. In the case the *twin edge* was not visited yet, you have to compare the cluster's normal with the normal of the triangle associated with the *twin edge*. If this test returns that the normal vectors are equal, the algorithm calls itself recursively with the *twin edge* as current edge - else the current edge and its *twin edge* are exterior edges. Consequently, the current edge has to be added to the exterior edge set of the cluster. Next, the procedure has to be repeated with the *next edge* as well as with the *previous edge* of the current edge. The whole algorithm has to be executed while there are edges left, which were not visited, yet.

Algorithm 4.1 - Generating Clusters

```
 1    method buildClusters()
 2    begin
 3      while not all edges of the polyhedron were visited do
 4        Cluster c = new Cluster
 5        add c to the polyhedron's set of clusters
 6        Edge e = arbitrary edge, which is not visited yet
 7        Vector3d n1 = the normal of the triangle associated with e
 8        set the cluster normal of c to the value of n1
 9        call recursivelyBuildCluster(e, c)
10    end

11    method recursivelyBuildCluster(Edge e, Cluster c)
12    begin
13      add e to the edge set of c
14      add e's triangle to the triangle set of c
15      mark e as visited
16      Edge twin = the twin edge of e
17      call checkEdge(e, twin, c)
18      Edge next = the next edge of e
19      call checkEdge(e, next, c)
20      Edge prev = the previous edge of e
21      call checkEdge(e, prev, c)
22    end

23    method checkEdge(Edge e1, Edge e2, Cluster c)
24    begin
25      if e2 was visited already then
26        if e2 is exterior edge then
27          add e1 to the exterior edge set of c
28      else
29        Vector3d n2 = the normal of the triangle associated with e2
30        if c's normal equals n2 then
31          call recursivelyBuildCluster(e2, c)
32        else
33          add e1 to the exterior edge set of c
34          mark e1 as exterior edge
35          mark e2 as exterior edge
36    end
```

4.3 Computing Intersection Segments

After the initialization of the cluster structure, the intersection segments have to be computed. As already mentioned, only line segments will be regarded - point segments will be neglected. In order to find all intersection segments, it is necessary to check each triangle of the first polyhedron against each triangle of the other polyhedron for intersection.

Hubbard suggests a fast pretest with the help of a bounding box test. The developed software components differ from this suggestion. Instead of this, the fast triangle-triangle intersection test by *Möller* [3] is used to check, if two triangles intersect. This test is adapted to the fact that coplanar faces can be neglected as mentioned earlier. Consequently, if the test detects that two triangles are coplanar, it will not be tested, whether they really intersect or not, because these triangles will not be regarded anyway. Furthermore, the signed distances from the triangle's vertices to the other triangle's plane computed by the triangle intersection test will be stored in order to use them later in the algorithm. The signed distances will be computed by inserting the respective vertex into the plane equation of the other triangle (*Figure 4.5*). If these distances are near zero (within a certain tolerance), the distances will be set to exactly zero. In this way, a tolerance has not to be regarded within the following steps. Moreover, the direction vector of the intersection line between the triangles' planes (also computed by the *Möller* pretest) can be stored, too. Thus, it is not necessary to compute it again later when it is needed.

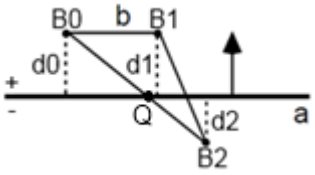


Figure 4.5 - Signed distances d0, d1 and d2 from the vertices of triangle *b* to the plane of triangle *a* (according to [1])

If the pretest detects triangles that intersect, an intersection line *L* between the triangles' normals has to be computed. The direction vector of the line is already known (computed by the *Möller* pretest via the cross product of the planes' normal vectors). To compute the location vector of the line, *Hubbard* suggests a method by *Segal and Sequin* [10] setting up a system of equations that allows finding the location vector without inverting a matrix. The location vector will be used later as a reference point on the line.

The implemented components use another way to find such a reference point. If the *Möller* pretest computes that a distance from a vertex of triangle *b* to *a*'s plane is zero, then the aligned vertex can be used as reference point, because this point is located in both planes and, consequently, on the intersection line. For example, if the distance *d2* is equal to zero, vertex *B2* can be used as reference point.

In the case no distance is equal to zero, the intersection point of *a*'s plane and an edge of *b* that cuts *a*'s plane can be used as reference point. An edge that cuts a plane can be detected by multiplying the signed distances of its end points. If this product is less than zero, the considered edge cuts the plane. For example, if *d0* is positive and *d2* is negative, then the product of *d0* and *d2* is less than zero. This means the edge between *B0* and *B2* cuts *a*'s plane. The exact point *Q* where *a*'s plane intersects the edge can be found easily, because the

distance from one of the edge's endpoint vertices to the plane of *b* is proportional to the distance along the edge from that vertex to Q:

$$\boldsymbol{Q} = \begin{cases} \boldsymbol{B0} + \dfrac{|d0|}{|d0| + |d1|} \cdot (\boldsymbol{B1} - \boldsymbol{B0}) & \mid d0 \cdot d1 < 0 \\ \boldsymbol{B1} + \dfrac{|d1|}{|d1| + |d2|} \cdot (\boldsymbol{B2} - \boldsymbol{B1}) & \mid d1 \cdot d2 < 0 \\ \boldsymbol{B2} + \dfrac{|d2|}{|d0| + |d2|} \cdot (\boldsymbol{B0} - \boldsymbol{B2}) & \mid d2 \cdot d0 < 0 \end{cases}$$

After the reference point is determined, the line segments along where one triangle intersects the other triangle's plane have to be found. Accordingly, two segments have to be calculated: *s1* and *s2* (*Figure 4.6*). The line segment *s1* arises by the intersection of *a* with the plane of *b*. On the other hand, the line segment *s2* arises by the intersection of *b* with the plane of *a*. Both segments will lie along the intersection line *L*.

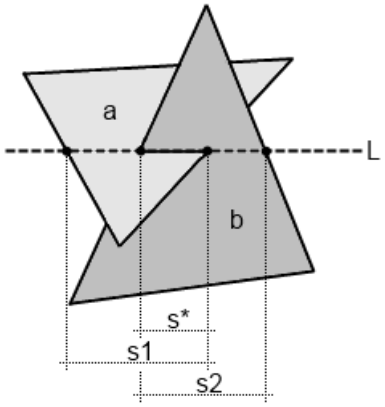


Figure 4.6 - Intersection segments *s1* and *s2* and their intersection *s

An endpoint of a segment can either be a triangle's vertex or a position on an edge of the current triangle. The vertices of the segment are computed in the same way as the reference point was computed. If the distance to the plane of the other triangle is equal to zero, then the aligned vertex is a vertex of the segment. If the distances of two end points of an edge have different signs then the intersection point *Q* can be computed as shown in the formulas above.

The segment's vertices are stored by the distance to the chosen reference point on the intersection line. In this way you can easily intersect these segments in the next step by comparing their distances on *L*. Intersecting *s1* and *s2*, a result *s** can be computed representing the final intersection segment between *a* and *b*. This may either be a line, a point or even empty. If *s** is not a line, the segment can be neglected. Else it should be connected to both of the polyhedral solids in order to use it later as a constraint for the cluster's triangulation.

If a segment's vertex is located on an *exterior edge*, this vertex has to be added to the edge's set of intermediate points as well as to the respective set of the edge's *twin edge*. This is necessary for the triangulation step to split these edges into two parts.

As soon *s** is computed, its end vertices will be added to the global coordinate map. In this manner, these vertices can be shared by another segment located at the same coordinate. Furthermore, the segment's vertices (or rather the corresponding indices) and the segment itself will be added to the point segment map of each polyhedron, which will be used later for optimizing the segments.

4.4 Optimizing Intersection Segments

The key to improve the performance of the original *Laidlaw et al.* algorithm is the more global splitting phase. Triangles are not split as soon as the intersection between them is discovered. Rather all splitting is postponed until all intersections have been found.

The constrained triangulation is performed using a cluster's exterior edges and intersection segments as constraints. This approach can still create extraneous triangles, since the segments are created by intersecting the triangles against each other instead of intersecting the cluster's polygonal boundary.

Figure 4.7 depicts an intersection of two clusters and shows that because of this approach several intersection segments will be created (*s1*, *s2*, *s3* and *s4*). Afterwards, extraneous triangles will be generated using these segments as constraints for the triangulation. *Figure 4.8* shows that reducing these segments will also reduce the number of generated triangles. Each cluster is then split into the minimum number of triangles needed to keep it from penetrating the other polyhedron. This brings up some advantages: Obviously, time and effort can be saved within the classification phase, if a fewer number of triangles have to be classified. Besides, a result solid with fewer triangles will also decrease the time and effort for other potential Boolean set operations that shall be executed after the current operation. As a consequence, the number of triangle intersections that has to be tested will be reduced.

It is necessary to combine the intersection segments as far as possible to reduce the number of generated triangles. Segments that can be combined are distinguished by the fact, that they are parallel and that they share a common vertex. The point segment map of a polyhedron is used to detect such segments located at a common vertex.

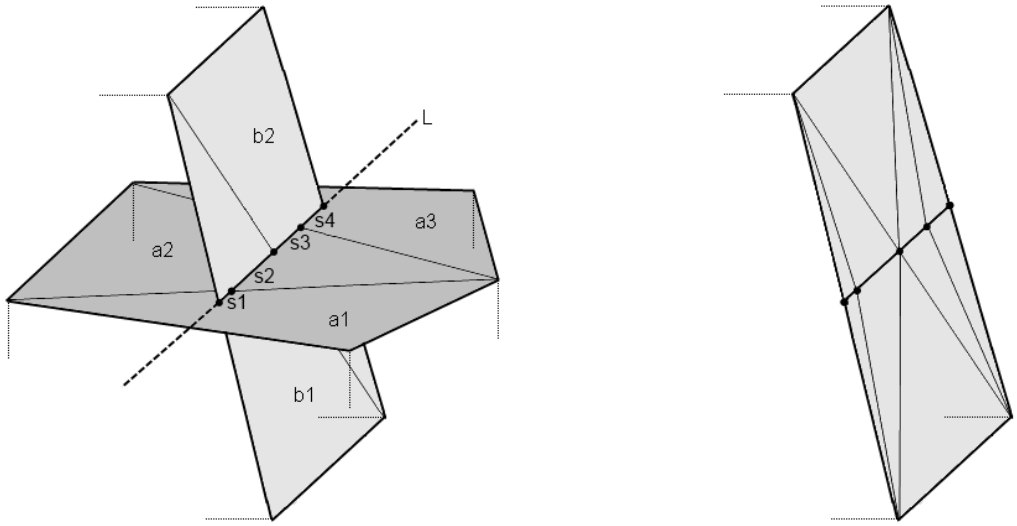


Figure 4.7 - The intersection of two clusters, and the result of cluster *b* splitted without reducing the intersection segments

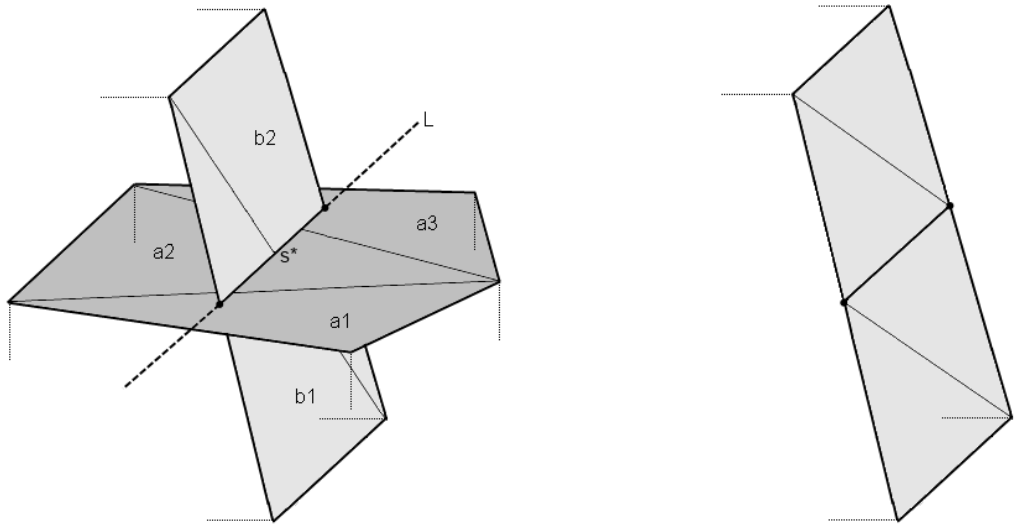


Figure 4.8 - The intersection of two clusters, and the result of cluster *b* splitted after reducing the intersection segments

Algorithm 4.2 investigates all segments of a cluster, whether they can be combined or not. It starts with an arbitrary segment and adds this segment to a set collecting all segments that can be combined. Next, the adjacent segments will be detected, which are located at the end vertices of the current segment. If there is more than one other segment, it is not possible to combine the segments at this endpoint, anyhow. For instance, this can occur at a corner of a solid, where three edges convene. Otherwise, the direction vectors of the segments have to be checked for parallelism. If they are parallel, this segment has to be added to the combination set, too. Now again, the adjacent segments of this second segment have to be investigated in the same way. This has to be repeated until no other parallel segment can be found.

Afterwards, the segments that can be combined will be united to one segment. Thereby, the minimum and maximum distance on the intersection line *L* will be computed to a reference

point on the line. After this, a segment with vertices at the minimum and maximum location will be created and the initial segments will be deleted.

It is necessary to compare the direction vector of the current segment with the direction vector of the first segment you choose to avoid that the algorithm combines segments on a curve (see also *chapter 4.2*).

The stack within *Algorithm 4.2* is used to process all detected segments successively. For example, assume the first arbitrary segment from *Figure 4.7* is *s2*, then *s2* is pushed on this stack and directly thereafter it is popped again to investigate it. The algorithm detects two adjacent, parallel segments (*s1* and *s3*) and pushes these two elements onto the stack. In the following step the next segment (*s3*) is popped from the stack. Another segment *s4* is found and it is pushed onto the stack. Now *s4* will be investigated, but no adjacent, parallel segment will be found - the same applies for the next element from the stack (*s1*). The stack is empty after *s1* is popped from it, consequently, the segments *s1*, *s2*, *s3* and *s4* can be combined in the next step to a segment *s**, which is illustrated in *Figure 4.8*.

Algorithm 4.2 - Optimizing Segments

```
 1    method reduceSegments()
 2    begin
 3      while not all segments of the cluster have been investigated do
 4        Segment start = arbitrary segment, which is not investigated yet
 5        Vector3d startDir = direction vector of start
 6        Stack<Segment> stack = new Stack<Segment>
 7        push start onto the stack
 8        Set<Segment> canBeCombinedSet = new Set<Segment>
 9        while stack is not empty do
10          Segment s1 = pop the next segment from the stack
11          mark s1 as investigated
12          add s1 to the canBeCombinedSet
13          foreach point p of s1 (start and end point) do
14            Set<Segment> x = all segments located at p
15            remove s1 from x
16            if size of x != 1 then
17              continue
18            Segment s2 = the remaining segment from x
19            if s2 was not investigated yet then
20              if startDir is parallel to the direction vector of s2 then
21                push s2 onto the stack
22        unite the segments from the canBeCombinedSet
23    end
```

4.5 Splitting Clusters by Triangulation

Once all intersections have been found and optimized, the splitting of the clusters along the detected intersections can be performed. The splitting is an instance of a constrained triangulation problem. The constraints are the cluster's exterior edges and the optimized intersection segments. Because the cluster's edges and segments all lie in a plane, the problem can be reduced to a planar triangulation problem.

The developed software components use a triangulator based on the algorithm described by *Preparata and Shamos* [9]. The triangulation algorithm is not part of this paper, but there is a visualization available under: *http://www.cfm.brown.edu/people/baolin* - for more details see also [4].

Each cluster that has to be splitted by intersection segments will be re-triangulated. The old triangle and edge objects will be deleted. Because the triangulator is limited to 2D coordinates, they have to be projected to a 2D plane. The coordinates will either be projected on the XY-plane, the YZ-plane or the XZ-plane. Thereby, no conversion has to be computed - just the corresponding x-, y- or z-values have to be extracted from the 3D coordinates. In this way, time is saved, because no additionally computation has to be performed. However, the geometry will be distorted using such a simple projection, but this does not matter, since just the information how to create the new triangles is important, and this is not deranged by the distortion. This is true as long as the original shape is not lost. *Figure 4.9* depicts a face and its projections to these planes. As you can see, it is possible that a face can be projected on a plane, whereas the original shape gets lost: The triangle shown in this figure is just a line projected to the XY-plane. In this extreme case it is no longer possible to perform a correct triangulation.

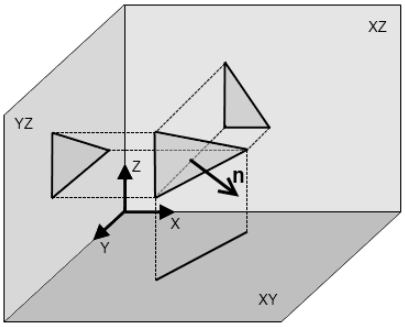


Figure 4.9 - A triangle and its projections

For this reason, it is important to decide, which of these three planes is the most suitable plane for projection. This can be detected by analyzing the normal vector *n* of the cluster. The biggest value of this vector decides on which plane has to be used for projection. For example, assume *n* to (1.0 / 0.0 / 0.0), then the most suitable plane is the YZ-plane, because in the other two planes the cluster would degenerate to a line. Consequently, the most suitable plane for projection results after the following scheme:

- **if** normal.x ≥ normal.y **and** normal.x ≥ normal.z **then** YZ-plane
- **if** normal.y ≥ normal.x **and** normal.y ≥ normal.z **then** XZ-plane
- **else** XY-plane

After a cluster is splitted by triangulation, too many triangles will be created, if the cluster had holes before the triangulation. This accrues, since the triangulator does not support holes

anyhow. The holes will also be meshed with triangles. Fortunately, it is very easy to detect triangles that are located in the area where the holes should be.

Figure 4.10 shows a cluster with a hole (bounded by the vertices *#5*, *#6*, *#7*, and *#8*). In the bottom right corner two intersection segments are implied by the dashed lines. After the triangulation, the cluster is splitted along these intersection segments, but, unfortunately, the hole is filled with triangles. Before the triangulation, we already knew, among other things, that the edge from vertex *#5* to *#6* is an *exterior edge* of the cluster separating the hole from the cluster's surface. Consequently, an edge with the inverse direction from *#6* to *#5* is not allowed after the triangulation. Thus, the triangle containing this edge has to be deleted (*triangle <6, 5, 8>*). In general, this means a triangle containing an edge with the *twin key* of a cluster's exterior edge can be dismissed after the triangulation. Afterwards, all direct and indirect neighbors within the boundary of the hole can be deleted, too.

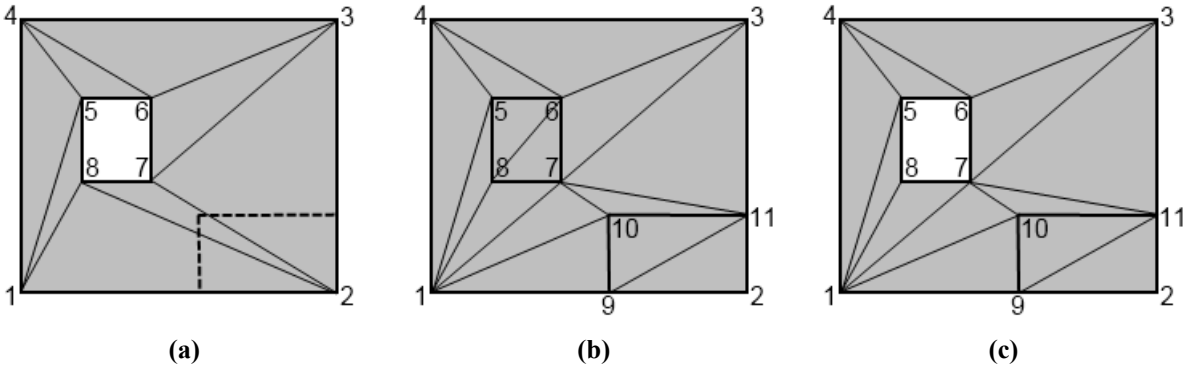


Figure 4.10 - A cluster with a hole and the computed intersection segments before the triangulation (a), after the triangulation before removing the hole triangles (b), and finally after removing the hole triangles (c)

4.6 Classifying Triangles

The classification phase of the *Laidlaw et al.* algorithm is based on ray casting. This procedure is time intensive and becomes the dominant factor in the overall run time in comparison to the improved splitting phase by *Hubbard*. Thus, *Hubbard* also introduces an improved classification phase by taking advantage of the restriction to triangular faces.

In the case intersection segments could be detected, the ray casting step is unnecessary. Instead of this, the triangles can be classified by analyzing the constellation of two triangles from one polyhedron relative to a triangle of the other polyhedron along a common edge, how it will be created along an intersection segment. The first solid touches the surface of the other solid at this point, whereas the classification changes.

The expense can be amortized by propagating the classification of a triangle to its neighbors in its polyhedron as soon as it is determined. The propagation can continue as long as it does not reach another intersection segment.

As mentioned earlier, ray casting is only used in the case where no intersection segment could be detected. This means that a polyhedron is placed fully inside or outside the other polyhedron. Thereby, one vertex from the first solid is used to classify its location relative to the other solid. If the ray casting test returns that the vertex is located inside the other solid, all triangles of the first solid can be classified as *inside*. The triangles from the other solid can be classified as *outside*. In the case the vertex is located outside, the test has to be repeated with a vertex from the other polyhedron in relation to the first polyhedron in order to detect, if the other solid is located fully inside the first solid. If both tests signalize that no polyhedron is fully inside the respective other polyhedron, all triangles of both solids can be classified as *outside*.

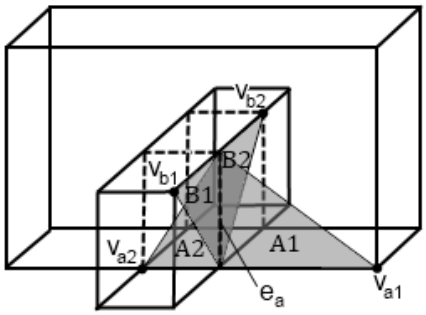


Figure 4.11 - Four Triangles along a common edge e_a, that connects two vertices of an intersection segment

The classification algorithm starts with an arbitrary edge e_a that connects two vertices of an intersection segment. This edge exists in the first polyhedron as well as in the other polyhedron, because the intersection segment was a common constraint for splitting the respective clusters (more precisely, not the edge objects itself exists in both polyhedra, rather two edge objects can be detected via the polyhedron's edge map of the respective polyhedron connecting the same vertices). At this edge four different triangles are located as illustrated in *Figure 4.11* - two in each polyhedron.

Assume *A1* should be classified; the first step is to detect the vertex v_{a1} of *A1* that is not located on the considered edge e_a as well as the triangles from the other polyhedron sharing this edge, too (*B1* and *B2*), and the vertices v_{b1} and v_{b2} (also not located on e_a like v_{a1}). Next, the signed distances from v_{a1} to the plane of both *B1* and *B2* have to be computed as well as the distance from v_{b1} to the plane of *B2*. These distances are named as follows:

- *a_b1*: the distance of v_{a1} to the plane of *B1*
- *a_b2*: the distance of v_{a1} to the plane of *B2*
- *b1_b2*: the distance of v_{b1} to the plane of *B2*

With the help of these distances the classification of *A1* can completely be determined using the algorithm shown in *Figure 4.12*. In order to avoid using a tolerance during this classification algorithm, the distances are set to exactly zero, if they are almost zero within a certain tolerance.

The classification depends on whether *B1* and *B2* locally define a convex or concave surface. If the distance *b1_b2* is less or equal than zero, *B1* and *B2* are convex. In this case *A1* is located *inside* the other polyhedron, if both distances *a_b1* and *a_b2* are less than zero. On the other hand, if they define a concave surface, it is sufficient to classify *A1* as *inside*, if either *a_b1* or *a_b2* is less than zero.

If one of these distances equals zero and the other distance is *less* or equal than zero (in the convex case) or if the other distance is *greater* or equal than zero (in the concave case), *A1* is located on the surface of the other polyhedron. Because of the determinations which of the coplanar faces belong in the result of the Boolean set operation (*same* or *opposite*), it is necessary to compare the normal vectors of the respective triangles. If *a_b1* equals zero the normals of *A1* and *B1* have to be compared. On the other hand, if *a_b2* equals zero the normals of *A1* and *B2* have to be compared. If the normals are pointing into the same half space, *A1* can be classified as *same*, otherwise as *opposite*. In all other cases *A1* is located *outside* the other polyhedron.

In reference to the example illustrated in *Figure 4.11*, the distances *a_b1* and *a_b2* are positive, because v_{a1} is located in the half space the triangle's normals of *B1* and *B2* are pointing into. The distance *b1_b2* is exactly zero, because *B1* and *B2* are coplanar faces. According to the presented algorithm, *A1* can be classified as *outside*. Repeating the algorithm to determine the classification status of the other triangles, *A2* can be classified as *inside*, *B1* as *outside*, and finally *B2* as *inside*. After that, the classification can continue at another arbitrary edge that connects two vertices of an intersection segment.

As soon as a triangle is classified, the propagation step can start. All adjacent triangles that are not separated by an intersection segment can be classified with the same status as the initial triangle (*Figure 4.13*). The classification phase can be aborted as soon as all triangles of both polyhedra are classified. For example, after *A1* is classified as *outside*, all triangles that are not separated by an intersection segment (implied by the dashed lines) will get the same status, which is illustrated in *Figure 4.13(b)*. After the propagation of *A1* is completed, *A2* may be classified as *inside*. As you can see in *Figure 4.13(c)* only one other triangle can be classified, since the propagation reaches the boundary defined by the intersection segments.

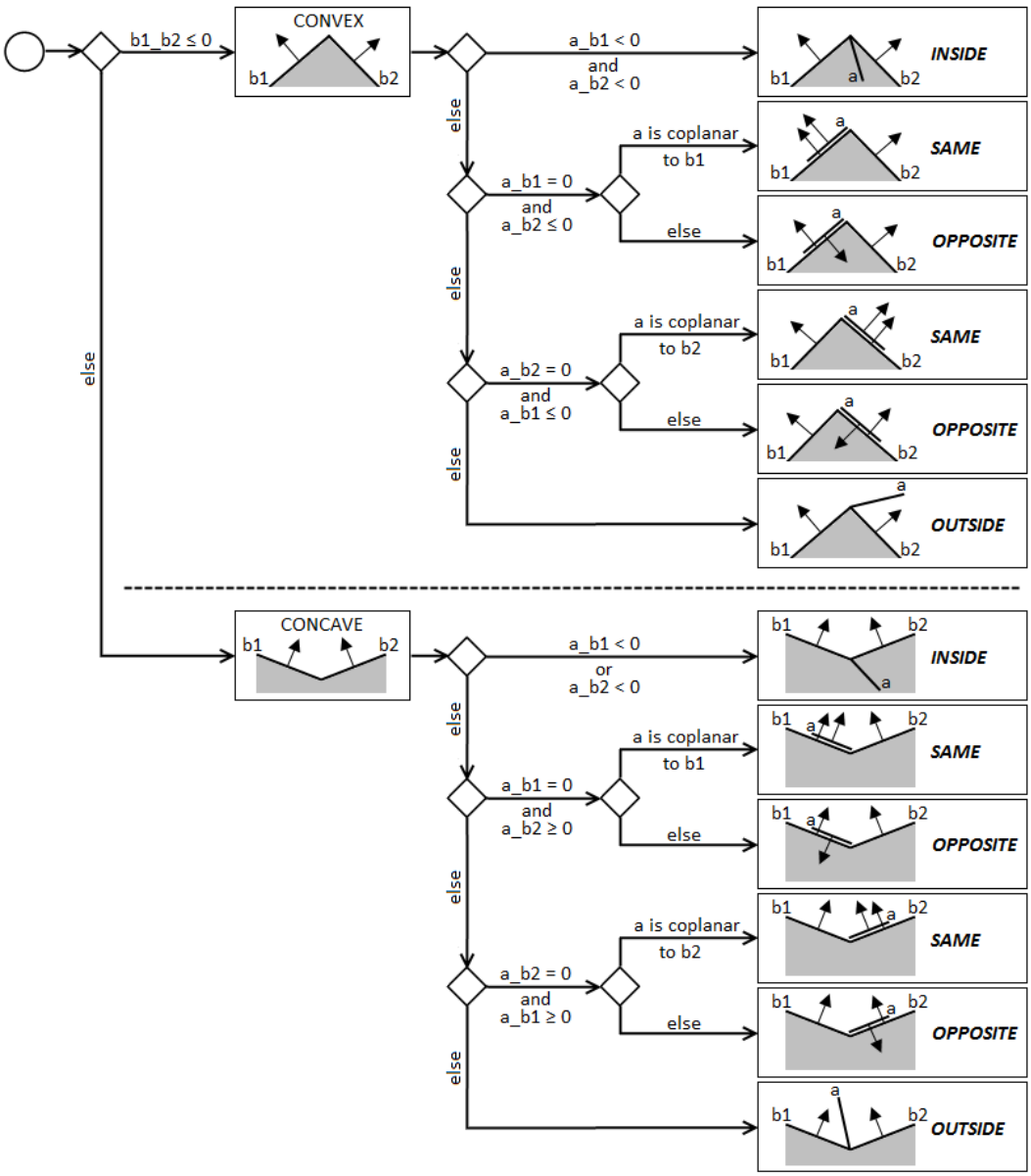
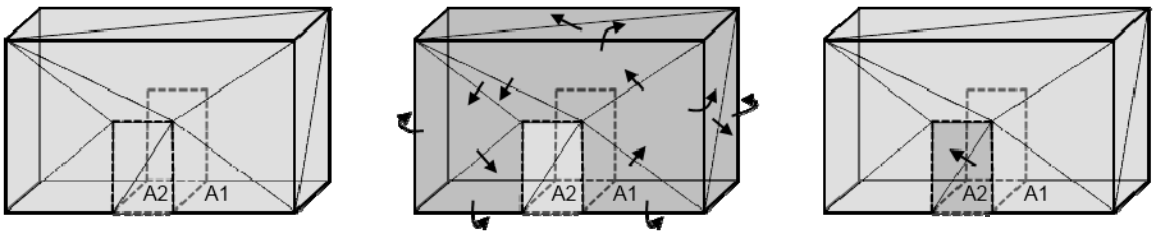


Figure 4.12 - Classification algorithm to compute the status of triangle a, the triangles b1 and b2 belong to the other polyhedron



(a) **(b)** **(c)**

Figure 4.13 - Example: Propagating the status of a triangle: situation after the splitting phase (a), propagation of A1's status (b), propagation of A2's status (c); for clarity the covered triangles have been hidden

4.7 Selecting Triangles

Once the faces of both polyhedra *A* and *B* have been classified, the result of the Boolean set operation can be created. *Figure 4.14* shows which triangles retain by each of the operations. The triangles from *B* that retain after taking the difference must have their orientations reversed, because the interior of *B* becomes the exterior of the Boolean difference of both. After retaining the necessary triangles, the result solid has to be created as it is described in *chapter 3.4*.

	Triangles from A				*Triangles from B*			
operation	*inside B*	*outside B*	*same*	*opposite*	*inside A*	*outside A*	*same*	*opposite*
A ∪ B	no	yes	yes	no	no	yes	no	no
A ∩ B	yes	no	yes	no	yes	no	no	no
A \ B	no	yes	no	yes	yes *	no	no	no

Figure 4.14 - The classifications of triangles that belong in the results of Boolean set operations [1]; (*…triangles with reverse orientations)

4.8 Simultaneous Computation of Boolean Set Operations with more than two Solids

The previous chapters depictured the algorithms to compute the result of a Boolean set operation of two solids *A* and *B*. However, building elements are often the result of more than one operation. For example, *Figure 4.15* shows a wall with a lot of window openings, which will be subtracted successively. Each subtraction requires that all the steps of presented algorithm have to be repeated again and again, which is very time intensive. In addition, in each step more triangles have to be tested for intersection.

Within this chapter an approach will be presented computing the result of several Boolean set operations simultaneously. Thereby, the result of a Boolean operation between a solid and a set of other solids called *secondary* solids will be computed. This algorithm reduces the execution time of problems as shown in *Figure 4.15* dramatically. Therefore, the algorithms presented in the previous chapters have just to be rearranged in its execution sequence.

The algorithm is restricted to *secondary* solids that do not intersect or touch each other. This is necessary, because otherwise too many special cases can occur within the algorithms that have to be caught. Furthermore, with this restriction only the Boolean difference (\) and union (∪) of several solids make sense, because the Boolean intersection (∩) of several solids, that do not intersect, is always empty.

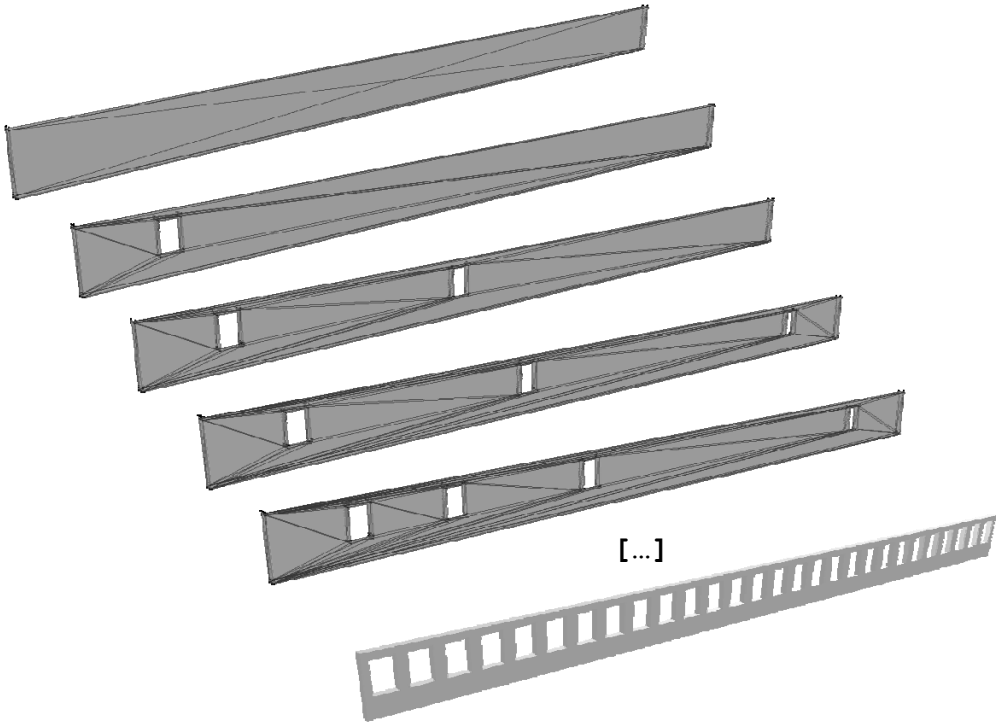


Figure 4.15 - Successive execution of Boolean Set Operations

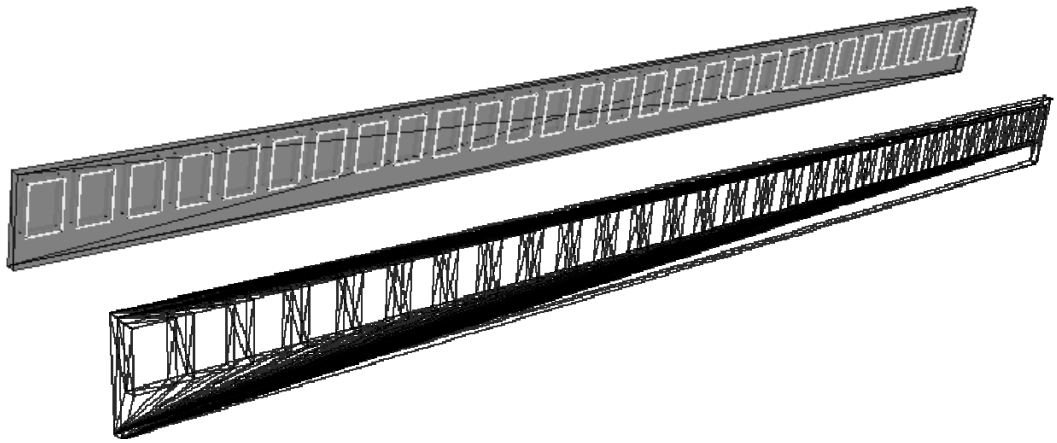


Figure 4.16 - Simultaneous execution of Boolean Set Operations

Algorithm 4.3 shows the slightly changed execution sequence of the algorithms to compute the result of a Boolean set operation of several solids simultaneously. The first step is to determine, if the algorithm is applicable in respect to the algorithm's restrictions, as mentioned earlier. It has to be detected, whether the *secondary* shapes intersect (touching included) or not. An efficient method to check this quickly is to perform a bounding box test. If this pretest signalizes an intersection, the *secondary* shapes have to be subtracted successively and the improved algorithm cannot been applied.

On the other hand, if it is applicable, the polyhedron structure has to be created for the first polyhedron as well as for all *secondary* polyhedra. In a loop over all *secondary* solids, the

intersection segments between the first solid and the current *secondary* solid have to be computed and optimized. Afterwards, still in this first loop, the current *secondary* solid has to be splitted by triangulation.

However, the triangulation of the first solid can be performed not before the first loop is completed, because all intersection segments with all the other solids have to be computed first. Once all intersection segments are found, the triangulation of the first solid can be performed in one single step using all of a cluster's intersection segments and of course its exterior edges as constraints (*Figure 4.16*).

Afterwards, the classification has to be performed by iterating over the *secondary* polyhedra again. Starting with an intersection segment of such a solid, the classification has to be performed, since all the triangles of this *secondary* polyhedron are classified as it is described in *chapter 4.6*. Thereby, the propagation can continue as long as it does not reach another intersection segment (also if this intersection segment is associated to another *secondary* solid). Finally the needed triangles, depending on which operation is performed, will be used to create the correct result of the Boolean set operation.

Algorithm 4.3 - Simultaneous Computation of Boolean Set Operations with more than Solids

```
 1    method getResultOfBooleanOperation(Shape3D s1, List<Shape3D> secondary)
 2    begin
 3       if secondary shapes intersect each other then
 4          perform the Boolean operation successively and return the result
 5
 6       Polyhedron p1 = convert s1 (Java3D > Hubbard, chapter 3.4)
 7       generate clusters for p1 (chapter 4.2)
 8       List<Polyhedron> polyList = new List<Polyhedron>
 9
10       foreach Shape3D s2 from secondary shapes list do
11          Polyhedron p2 = convert s2 (Java3D > Hubbard, chapter 3.4)
12          generate clusters for p2 (chapter 4.2)
13          add p2 to polyList
14          compute intersection segments between p1 and p2 (chapter 4.3)
15          optimize intersection segments (chapter 4.4)
16          split clusters of p2 by triangulation (chapter 4.5)
17
18       split clusters of p1 by triangulation (chapter 4.5)
19       Set<Triangle> result = new Set<Triangle>
20
21       foreach Polyhedron p2 from the polyList do
22          classify triangles of p1 and p2 (chapter 4.6)
23          select needed triangles and add them to the result (chapter 4.7)
24
25       convert the result (Hubbard > Java3D, chapter 3.4)
26       return the converted result
27    end
```

4.9 Clipping

This section shows the adaption of the presented algorithms to compute the result of a solid that has to be clipped at a specified plane. This operation is much easier than a Boolean set operation on two solids by the fact that only one plane is checked against the solid's triangles.

The faces have to be classified as *below* or *above* the plane, whereby *above* means the half space, the plane's normal points into. Triangles that are located within the plane will not be regarded, because the faces, which will arise in the plane, will be created by triangulating the intersection segment constraints located in the plane. Faces that are located partially *above* and *below* the plane have to be splitted by triangulation. The result of the clipping procedure will contain all triangles *below* the plane as well as all triangles located in the plane. This is just an assignment and can also be defined reverse.

The algorithm starts with a quick pretest excluding the cases, where a solid is completely *above* or *below* the specified plane. Thereby, the signed distances from all of the solid's vertices to the plane will be computed (see *chapter 4.3*). If all distances are less or equal than zero, the whole solid has to be returned, because the solid is then located completely *below* the specified plane. In the opposed case, whereas all distances are greater or equal than zero, the result set is empty.

If these special cases can be excluded as described above, there might be faces that are located partially above and below the plane. All triangles of the solid have now to be checked against the plane. If an intersection is detected, an intersection segment has to be created analogous to the algorithm from *chapter 4.3*. This segment has to be associated to both plane and current cluster.

Once all intersection segments have been computed, they have to be optimized in the same way as depicted in *chapter 4.4*. Afterwards, the clusters will be splitted by triangulation - see chapter 4.5. Additionally, the plane will be triangulated using the computed intersection segments in order to limit the area that should be triangulated.

As soon as all triangles are splitted, the result solid can be created like it is described in *chapter 3.4* using the triangles *below* the plane as well as the triangles from the plane's triangulation.

Polygonal Bounded Clipping:

Polygonal bounded clipping is currently not based on the clipping algorithm as described above. Instead, the in one direction unbounded half space (see *Figure 2.5*) will be bounded by computing the dimensions of the solid that shall be clipped. Afterwards, the result will be computed by taking the Boolean difference of the solid and the computed (now in all directions bounded) half space solid.

5 Error Discussion

5.1 Detected Software Bugs

During the creation of this student research project, several mistakes in the implementation have been found. Some of these errors led to serious changes in the source code and have also been integrated into the algorithms' depictions within this paper.

The change with the most positive effect in the light of minimizing errors was the introduction of a global coordinate map for all polyhedra as described in *chapter 3.2*. Previously, it was tested very intricately and erroneous, whether a vertex already exists within a certain tolerance or not. Hence, vertices were created twice, which led to incompatible triangle meshes, since the edges adjoined at these doubly existing vertices were not using the same vertex indices as illustrated in *Figure 5.1*. As a consequence, the program crashed at a later date when the classification should be propagated, because some *twin edges* did not exist in such incompatible triangle meshes. This led to *null pointer exceptions* while searching for a specific *twin edge*, because the *twin edge's* key was not found within the polyhedron's map of edges. Certainly, these exceptions must not happen, since every edge must have a *twin edge* within a regular 2-manifold solid.

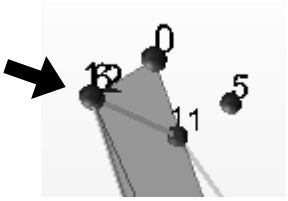


Figure 5.1 - Bug while searching equal vertices: The program did not found the already existing vertex #6, hence, a new vertex #12 was created. The adjacent triangle meshes based on the vertex index structure became incompatible, which led to a crash at a later date. The introduction of the global coordinate map solves the problem.

Another bug could be detected while projecting vertices into a plane as described in *chapter 4.5*. This method brings up a distortion of the triangle mesh structure. As a consequence, the tolerance has to be distorted in the same way, because otherwise it is inconsistent in comparison to the distorted geometry. This was ignored before and might lead to inconsistent states. This problem can be solved by computing the angle α between the cluster's spanned plane and the projection's plane. Next, the tolerance *TOL* has to be multiplied with the cosine of α to get the distorted tolerance *TOL** as illustrated in *Figure 5.2*. Afterwards, this distorted tolerance can be used within the triangulator for comparing vertices.

Further, the program used the tolerance that is specified in the loaded *IFC* model. This is problematic, because a too small tolerance may produce results that do not represent the designer's intention. An example for this problem is illustrated in *Figure 5.3*. If the tolerance

becomes too small, a stairway opening is not "punched" through a ceiling. An infinitesimal piece remains, because the upper vertices of the subtraction solid are classified as within the ceiling instead of on its surface. The inaccuracies arise by floating point arithmetic errors, which will be enforced by using lots of local coordinate system transformations as it is used in the *Java3D scene graph* as well as within the *IFC* model. This problem can be avoided using a fixed tolerance. A tolerance of *1.0e-5* Meter has proved after testing various building models. Further strategies to solve numerical inaccuracy problems in solid modeling are discussed in the next chapter. In general, the number of transformations should be minimized in order to eliminate numerical inaccuracies by floating point operations.

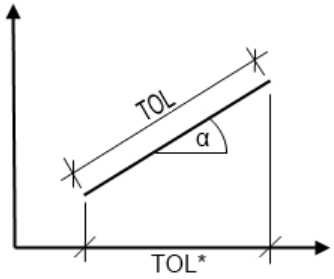
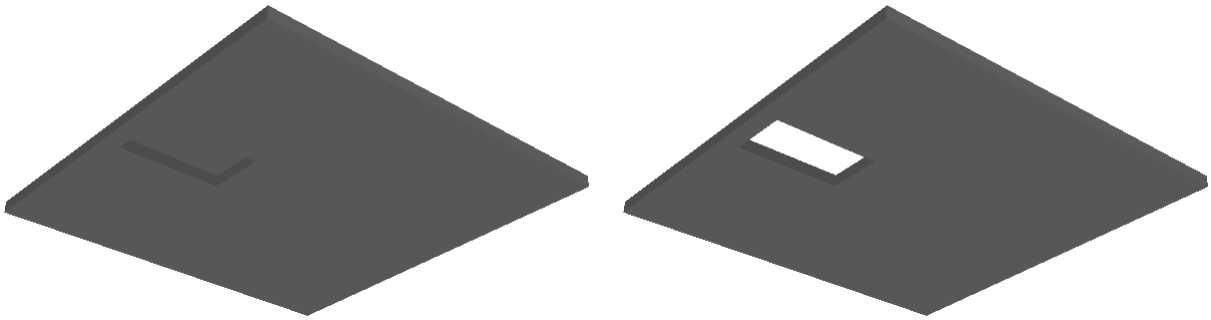


Figure 5.2 - The tolerance TOL and the adapted tolerance TOL*



(a) **(b)**

Figure 5.3 - Problems with too small tolerances: An opening for a stairway is not "punched" through a ceiling using a tolerance of 1e-10m (a), with a bigger tolerance of 1e-5m it looks fine (b)

Further bugs could be detected, which will just be sketched briefly:

- A whole partial algorithm to sort the exterior edges of a cluster was removed, because the triangulator does not need the exterior edges in a specific sequence. The edges are now added in an arbitrary sequence.
- Triangles that are located within the boundary of a cluster's hole, but not containing an edge with a *twin key* of a cluster's *exterior edge*, will now be neglected, too.
- The triangulation of a cluster will not be executed, if no intersection segment was found. Instead of this, the old triangles and edges can be used. This will reduce the execution time.

Another problem that has to be treated appears while converting the *Java3D* model to the extended *Hubbard* model. The indexed geometry of the *Java3D* model may contain wrong initial data. Not every triangle you get from the *coordinate index array* is a valid triangle. Sometimes you get nonsense like a triangle from point *A* to *B* and back to *A* or something like that. A wrong handling with the *Java3D* classes to create the geometry was not found in the source code of the developed components. Hence, this is probably a bug of the triangulator of the current *Java3D* version (v1.52). Anyway, those triangles can be neglected without any loss of information.

5.2 Restrictions of the implemented Boolean Algorithms

Restrictions: The implemented Boolean modeler is currently restricted to 2-manifold solids. Multiple coincident edges are not allowed, since each edge is registered by its name, that is composed by the indices of the connected vertices. If a coincident edge will be added, the mapping of the first edge would be overwritten in the polyhedron's edge map and the program will crash at a later date, because this case is not scheduled.

The initial data should also contain only solids with a closed shell - else the program will crash, because an open shell contains *exterior edges* that have no *twin edges*, which is also not scheduled in the program sequence, as mentioned earlier. In general, solids with an open shell are not suited for Boolean set operations, because the volume of the solid is not bounded completely.

Validity checks: The edge map of a polyhedron offers an efficient possibility to check the correctness of the input data. A key of an edge must be unique and should not be putted twice to this resource. If this will happen contrary to expectations, a coincident edge is detected, which is not supported, as already mentioned.

Another validation check may verify the existence of the present keys and their corresponding *twin keys*. If this test detects a key whose twin key is not contained in the polyhedron's edge map, some triangles are missing to build a closed shell.

In both cases, the Boolean set operation has to be aborted with an appropriate error message. In the same way, the edge map can also be used to control the results of the triangulation after the splitting phase.

5.3 Problems of Accuracy and Robustness

In geometric algorithms, as the presented Boolean modeler algorithm, decisions are based on the result of numerical computations of geometric relations between objects (vertices, edges and planes). Because of the imperfect representation of floating point numbers in a computer, tolerances have to be introduced. Unfortunately, using tolerances may result in inconsistent

decisions. For example, imagine three points P_1, P_2 and P_3 that are located in a close proximity (*Figure 5.4*). Using the coordinate map introduced in *chapter 3.2* in order to shift vertices on top of each other, the result differs in dependency of the adding sequence of these points. If P_1 is added first followed by P_2 then P_2 will be shifted to P_1. In the case P_3 is added first followed by P_2 then P_2 will be shifted to P_3. Otherwise if P_2 is added first followed by P_1 and P_3, all three vertices will be detected as coincident and will be shifted to the position of P_2. As you can see, the adding sequence of these points will have a big influence on how points are classified and shifted. Thus, faces using these points will slightly change its position and its surface normal in dependency of how the vertices are shifted. This arbitrary adding of points will have a bad influence to the robustness of the algorithm, because in some cases the algorithm will not produce an error, in other cases it fails.

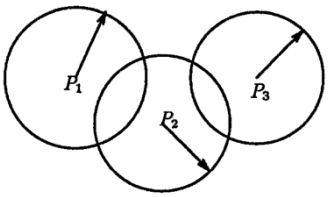


Figure 5.4 - Three points in a close proximity [12]

In technical literature several approaches are developed to increase the robustness of solid modeling algorithms. In the following, an overview presents the different kinds of approaches as well as an assessment, how far these approaches are applicable for the presented Boolean modeling algorithm from this paper.

Tolerance-based approaches: A tolerance-based approach, as it is used in the presented algorithms, takes into account that floating point numbers only have a limited accuracy. For this reason, each decision has to be validated using interval arithmetic. In this paper a fixed tolerance is used to compare vertices and vectors.

Other approaches introduce a non fixed tolerance that is associated with these objects. The tolerance or rather the error is updated according to the decisions made in order to maintain the consistency of the decisions [13, 14].

Based on these approaches *Fang et al.* [12] describe a method, where tolerances are also dynamically updated to preserve the theoretical properties of the relations. Additionally, they introduce a third relation *ambiguity* besides *coincidence* and *apartness* based on intuitionistic logic in order to obtain a consistent definition of geometric relationships. They promise a general approach "capable of handling different error types, such as numerical, arithmetic, approximation, design and manufacturing errors in a uniform way" (Fang et al. [12], p.4).

The named approaches are realizable within the implementation of the Boolean modeler. However, large changes have to be implemented, since dynamic tolerances are not intended anyhow. The approach of *Fang et al.* requires even more adoptions, because the relations

between objects have to be stored. Consequently, a decision cannot be made instantly. Rather decisions have to be seen in context to other decisions.

Precise computation: In several approaches precise computation is applied [15, 16, 17, 18], whereby exact numbers are used (for instance, exact algebraic numbers, space grids, bounded or unbounded rational numbers). The robustness of the algorithm is guaranteed, because no numerical error is introduced.

These approaches are not applicable for the presented Boolean algorithm, since the initial data given by the *IFC* building model is already not precise. As mentioned earlier, building elements are given within several local coordinate systems. The global position of an element has to be computed by multiplying the transform matrices of each local coordinate system. For this reason, a certain tolerance is needed to compensate the numerical error arising by floating point arithmetic operations.

Perturbation method: Other approaches use a perturbation method, whereby the input data is slightly changed in order to avoid positional degeneracy [11, 19, 20]. This technique should relieve the programmer from the task to provide a consistent treatment for every special case that can occur.

The term positional degeneracy denotes special cases and cannot be defined in general; its definition depends on the primitive operations used to solve the problem. For example, *Edelsbrunner and Mücke* [19] picture that term with reference to a point-in-polygon test. Thereby, a half-line that starts at the test point is checked for intersection with a polygon. If the number of intersections is odd, then the test point lies within the polygon. Assuming the test point is not located on the polygon's boundary, they differ between six cases that can occur. Four of them are denoted as degenerated, because the half line contains one or both endpoints of a polygon's line segment. Using now a perturbation method should exclude these degenerate cases.

A similar approach is used in the implemented ray casting step of the implemented Boolean modeler to classify, whether a vertex is located inside a polyhedron or not - see also *chapter 4.6*. However, not the initial data is slightly changed; rather the half-line's direction is perturbed, if a special case is detected.

In general, an advantage using this technique within the *Hubbard* algorithm is not recognizable at all, although it is often described as a "miracle cure" against robustness problems in geometric algorithms.

Scaling coordinate space: Floating point numbers in *Java* are built according to the international norm *IEEE 754*. Very small numbers (near zero) can be pictured as well as very large numbers. However, with increasing the number, the accuracy will become worse. This is the result of the compromise to picture a wide range of numbers.

Thus, *Firmenich* [21] scales the coordinate space to a range from -1 to +1 with the aim to reach a continuous accuracy over the whole coordinate space. The real coordinates are mapped to this scaled coordinate space (and vice versa) via transform matrices. This approach guarantees the user a constant accuracy and reduces numerical errors that arise by floating point arithmetic operations - especially if very large numbers are used.

If you think to building models, the algorithm has to deal with values normally less than 100.0 (assuming a building is normally not bigger than 100 Meter). In this range numerical inaccuracies are not due to the limited accuracy of floating point numbers, since we consider only a limited number of decimal places depending on the used tolerance. Using the standard tolerance of *1.0e-5* Meter, the accuracy of a 64-bit floating point number should be sufficient within this bounded domain. All in all, the implementation of this scaling to the Boolean modeler algorithm would have only a very limited influence to the avoidance of inaccuracies.

Shifting vertices of coplanar faces into a common plane: At the end of this chapter an example of a robustness problem will be shown that could be detected during the writing of this paper. This problem leads to an own approach, which should increase the robustness of the algorithm. The approach is not implemented yet. Consequently, a statement to its influence to the algorithm's robustness is just based on reflections.

The following problem arises, if two solids share a common edge, as illustrated in *Figure 5.5*. Both solids have an edge that passes the vertices *#8* and *#11*. By intersecting the non coplanar triangles (*a1*, *a2*, *b1* and *b2*) along these edge, two intersection segments *s** normally will be created, that are coincident.

However, in certain situations the algorithm creates only a degenerated variant of *s** - see *Figure 5.5(a)* - as well as the expected result of *s** - see *Figure 5.5(d)*. This happens, because the vertices of triangle *b1* deviates only a little bit from the plane of *a2*. This little deviation will have the effect that the vertices *#0* and *#3* will be classified as not within the plane ε_{b1} or more precisely within its surrounding tolerance - see *Figure 5.5(c)*.

According to the presented algorithm, an intersection segment *s2* will be created, which starts on the edge between vertex *#2* and *#0* and ends on the edge between *#0* and *#3*. The corresponding segment *s1*, which is created by the intersection of *b1* with ε_{a1} - see *Figure 5.5(b)*, starts at *#8* and ends at *#11*. The intersection of both segments *s1* and *s2* will produce the degenerated *s** as illustrated in *Figure 5.5(a)*.

On the right hand side this effect will not appear, since the vertices *#8* and *#11* will be barely classified as inside ε_{a2} - see *Figure 5.5(f)*. Consequently, both segments *s1* and *s2* will connect *#8* and *#11*, wherefore *s** is connecting these vertices, too, as illustrated in *Figure 5.5(d)*. All in all, two inconsistent segments *s** are created, that should not happen anyway.

In order to avoid such problems, all faces of both polyhedra shall be tested against each other and if two coplanar faces are detected, all vertices of both faces shall be transferred exactly into a common plane. In this manner the situation can be excluded that with an increasing distance between the triangle's endpoints the affiliation to a plane gets lost as it is illustrated in *Figure 5.5(c)*.

For example, in a pretest it will be detected that *b1* and *a2* are coplanar. Now, the vertices of one triangle should be transferred exactly into the plane of the other. The most suitable vertices for doing this are the vertices of *b1* (*Figure 5.6*), because if you transfer the vertices of the bigger triangle *a2*, the deviations will become larger and the original design intention could be lost. With the help of these corrections, the mentioned problem should no longer appear.

In principle, this approach follows the inverse strategy with regard to the perturbation method approach and tries to bring all vertices of coplanar faces into the same plane.

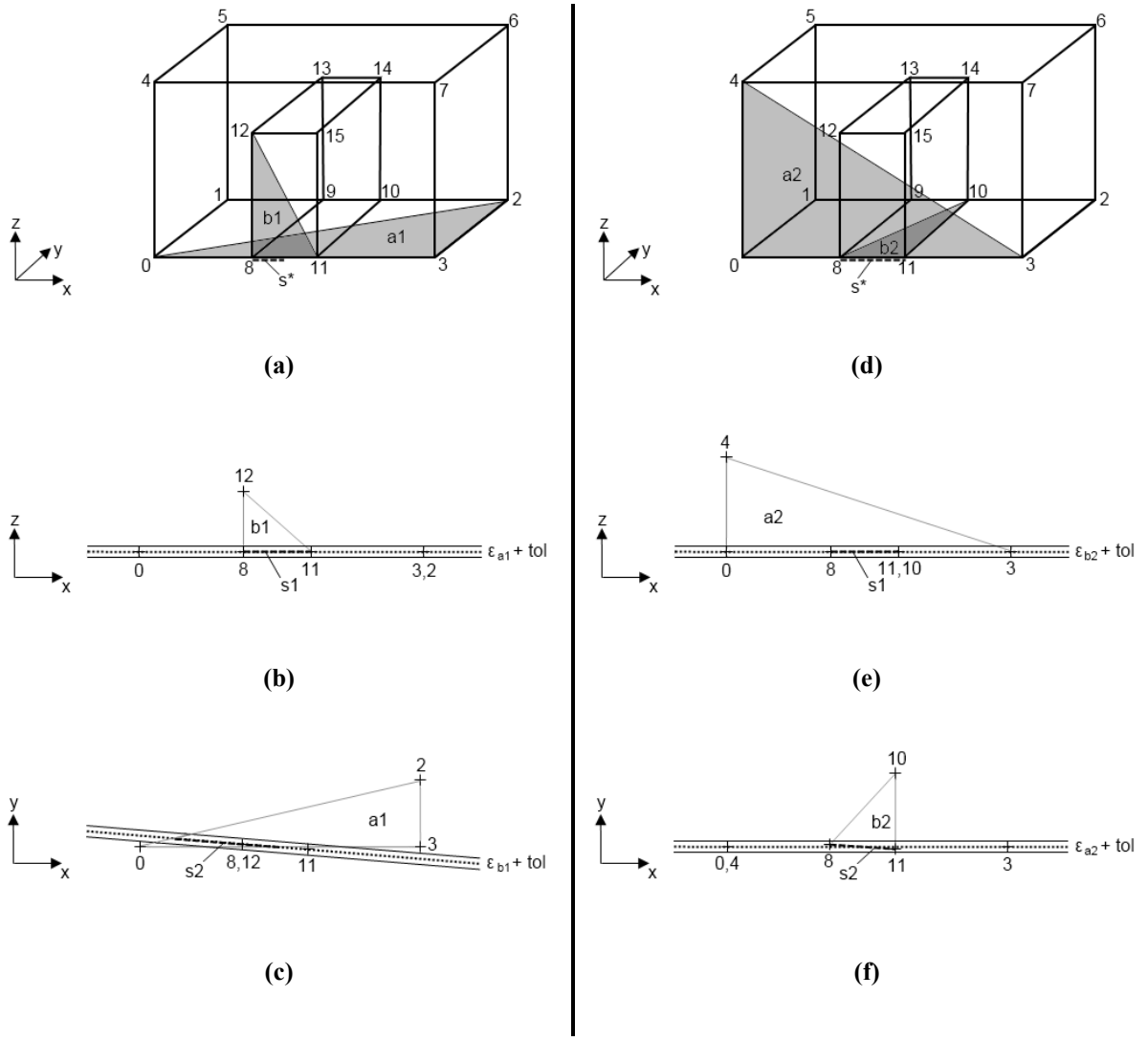


Figure 5.5 - Detected robustness problem that creates inconsistent segments *s

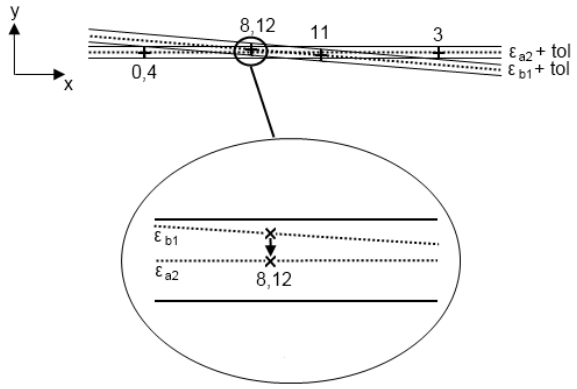


Figure 5.6 - Transferring the vertices of *b1* exactly into the plane of *a2*

6 Conclusion and Outlook

In order to document the implemented Boolean modeler, the algorithms have been reprocessed. Differences and improvements in comparison to the underlying *Hubbard* algorithm were presented. In addition, an extension was introduced that allows computing Boolean set operations with more than two solids simultaneously. Further, it was demonstrated how the *Hubbard* algorithm can be used to implement a clipping algorithm.

Within the fifth chapter, implementation errors were discussed that could be detected during the creation process of this work. Strategies to solve those bugs were shown. In addition, the restrictions of the implemented algorithms were outlined and it was shown how the initial data can be validated in order to check, whether the algorithm supports the given solid within its restrictions or not.

Summarizing the robustness problem, it was, as expected, not found *the one* approach that will solve *all* the robustness problems that could occur in the current implementation. Rather, some approaches were found in technical literature that will reduce special problems. Other approaches are not applicable with respect to the presented Boolean algorithms.

The next step to improve the algorithms' robustness shall be the implementation of the approach to shift vertices of coplanar faces into a common plane. In this way, the influence of infinitesimal deviations will be reduced.

Farther, the implementation of dynamically updated tolerances may reduce the number of inconsistent decisions that will be made. The approach of *Fang et al.* is not practical, since this would mean that the whole algorithm has to be written again following intuitionistic logic.

The implementation of the approach by *Firmenich* scaling the coordinate space would combat the problems that arise by numerical arithmetic errors. As shown, this will only have a limited influence to the robustness, since the accuracy of a 64-bit floating point number should be sufficient with respect to the used coordinate domain.

References

[1] Hubbard, P.M. (1990). *„Constructive Solid Geometry for Triangulated Polyhedra“*, Department of Computer Science, Brown University, Providence, Rhode Island 02912, CS-90-07.

[2] Laidlaw, D.H.; Trumbore, W.B.; Hughes J.F. (1986). *“Constructive Solid Geometry for Polyhedral Objects”*, Proceedings of SIGGRAPH ’86, published as Computer Graphics, Vol. 2, ACM, New York, USA.

[3] Möller, T. (1997). *"A Fast Triangle-Triangle Intersection Test"*, Journal of Graphics Tools, 2 (2), pp. 25-30.

[4] Tulke, J. (2010). *"Kollaborative Terminplanung auf Basis von Bauwerks-informationsmodellen"*, Dissertation, Bauhaus-Universität Weimar, Fakultät Bauingenieurwesen.

[5] *http://knol.google.com/k/3d-graphics*, last call: 2010-11-03

[6] Liebich, T. (2009). *"IFC 2x Edition 3 Model implementation Guide, Version 2.0, May 18, 2009"*, buildingSMART International Modeling Support Group.

[7] Bouvier, D.J. (1999). *"Getting Started with the Java3D API - A Tutorial for Beginners"*, Tutorial version 1.5, K Computing.

[8] Muller, D.E.; Preparata, F.P. (1978). *"Finding the intersection of two convex polyhedra"*, Theoret. Comput. Sci., 7:217-236.

[9] Preparata, F.P.; Shamos, M.I. (1985). *"Computational Geometry: An Introduction"*, Springer-Verlag, New York, USA.

[10] Segal, M.; Sequin, C.H. (1988). *"Partioning Polyhedral Objects into Nonintersecting Parts"*, IEEE Computer Graphics and Applications, January 1988, pp. 53-67.

[11] Ernst, P.H. (1995). *“Boolean Set Operations with Solid Models”*, Hewlett-Packard Journal, pp. 74-79, October 1995.

[12] Fang, S.; Brüderlin, B.; Zhu, X. (1993). *"Robustness in Solid Modeling - A Tolerance Based Intuitionistic Approach"*, Computer-Aided Design, 25(9):567-576, September 1993.

[13] Brüderlin, B. (1991). *"Robust regularized set operations on polyhedra"*, In Proc. of Hawaii International Conference on System Science, January 1991.

[14] Segal, M. (1990), *"Using tolerances to guarantee valid polyhedral modeling results"*, Computer Graphics 24, 4, pp. 105-114.

[15] Green, D.; Yao, F. (1986) *"Finite resolution computational geometry"*, In Proc. 27th IEEE Symp. Foundations of Computer Science, pp. 143-152.

[16] Ottmann, T; Thiemt, G.; Ullrich, C. (1987). *"Numerical stability of geometric algorithms"*, In ACM Annual Symposium on Computational Geometry (June 1987), pp. 119-125.

[17] Sughihara, K.; Iri, M. (1988). *"Geometric algorithms in finite precision arithmetic"*, Res. Mem. 88-14, Math. Eng. and Information Physicas, University of Tokyo.

[18] Sughihara, K.; Iri, M. (1989). *"A solid modeling system free from topological inconsistency"*, Journal of Information Processing 12, 4, pp. 380-393.

[19] Edelsbrunner, H.; Mücke, E. (1988). *"Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms"*, In Proc. of 4th ACM Symposium on Comp. Geometry (June 1988), pp. 118-133.

[20] Yap, C.K. (1988). *"A geometric consistency theorem for a symbolic perturbation theorem"*, In Proc. of 4th ACM Symposium on Comp. Geometry (June 1988), pp. 134-142.

[21] Firmenich, B. (2006). *"Vorlesungen über CAD in der Bauinformatik: Grundlagenorientierter Systementwurf"*, Script, Bauhaus-Universität Weimar, Fakultät Bauingenieurwesen.

[22] Selman, D. (2002). *"Java3D Programming"*. Manning Publications.

Figures

Algorithms

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

_____

Weimar, den 05.Mai 2010